Vulkan® 1.1.177 - A Specification (with all registered Vulkan extensions)

The Khronos® Vulkan Working Group

# Table of Contents

1. Preamble .................................................................................................................. 1  
2. Introduction .............................................................................................................. 3  
   2.1. Document Conventions ....................................................................................... 3  
3. Fundamentals .......................................................................................................... 6  
   3.1. Host and Device Environment ........................................................................... 6  
   3.2. Execution Model ............................................................................................... 6  
   3.3. Object Model .................................................................................................... 8  
   3.4. Application Binary Interface ............................................................................ 12  
   3.5. Command Syntax and Duration ....................................................................... 13  
   3.6. Threading Behavior ......................................................................................... 15  
   3.7. Errors .............................................................................................................. 29  
   3.8. Numeric Representation and Computation ....................................................... 68  
   3.9. Fixed-Point Data Conversions ......................................................................... 70  
   3.10. Common Object Types .................................................................................. 71  
4. Initialization ........................................................................................................... 74  
   4.1. Command Function Pointers ........................................................................... 74  
   4.2. Instances ........................................................................................................ 77  
5. Devices and Queues ................................................................................................ 89  
   5.1. Physical Devices .............................................................................................. 89  
   5.2. Devices .......................................................................................................... 113  
   5.3. Queues .......................................................................................................... 132  
6. Command Buffers .................................................................................................. 141  
   6.1. Command Buffer Lifecycle ............................................................................ 141  
   6.2. Command Pools .............................................................................................. 143  
   6.3. Command Buffer Allocation and Management ............................................ 151  
   6.4. Command Buffer Recording .......................................................................... 155  
   6.5. Command Buffer Submission ........................................................................ 166  
   6.6. Queue Forward Progress ............................................................................... 193  
   6.7. Secondary Command Buffer Execution ....................................................... 194  
   6.8. Command Buffer Device Mask ....................................................................... 198  
7. Synchronization and Cache Control ...................................................................... 201  
   7.1. Execution and Memory Dependencies ........................................................... 201  
   7.2. Implicit Synchronization Guarantees .............................................................. 230  
   7.3. Fences ............................................................................................................ 232  
   7.4. Semaphores .................................................................................................... 259  
   7.5. Events ............................................................................................................ 294  
   7.6. Pipeline Barriers ............................................................................................ 324  
   7.7. Memory Barriers ............................................................................................ 333
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.8. Wait Idle Operations</td>
<td>376</td>
</tr>
<tr>
<td>7.9. Host Write Ordering Guarantees</td>
<td>378</td>
</tr>
<tr>
<td>7.10. Synchronization and Multiple Physical Devices</td>
<td>378</td>
</tr>
<tr>
<td>7.11. Calibrated timestamps</td>
<td>378</td>
</tr>
<tr>
<td>8. Render Pass</td>
<td>383</td>
</tr>
<tr>
<td>8.1. Render Pass Creation</td>
<td>384</td>
</tr>
<tr>
<td>8.2. Render Pass Compatibility</td>
<td>445</td>
</tr>
<tr>
<td>8.3. Framebuffers</td>
<td>446</td>
</tr>
<tr>
<td>8.4. Render Pass Commands</td>
<td>459</td>
</tr>
<tr>
<td>9. Shaders</td>
<td>486</td>
</tr>
<tr>
<td>9.1. Shader Modules</td>
<td>486</td>
</tr>
<tr>
<td>9.2. Shader Execution</td>
<td>491</td>
</tr>
<tr>
<td>9.3. Shader Memory Access Ordering</td>
<td>491</td>
</tr>
<tr>
<td>9.4. Shader Inputs and Outputs</td>
<td>492</td>
</tr>
<tr>
<td>9.5. Task Shaders</td>
<td>492</td>
</tr>
<tr>
<td>9.6. Mesh Shaders</td>
<td>493</td>
</tr>
<tr>
<td>9.7. Vertex Shaders</td>
<td>493</td>
</tr>
<tr>
<td>9.8. Tessellation Control Shaders</td>
<td>494</td>
</tr>
<tr>
<td>9.9. Tessellation Evaluation Shaders</td>
<td>496</td>
</tr>
<tr>
<td>9.10. Geometry Shaders</td>
<td>496</td>
</tr>
<tr>
<td>9.11. Fragment Shaders</td>
<td>496</td>
</tr>
<tr>
<td>9.12. Compute Shaders</td>
<td>498</td>
</tr>
<tr>
<td>9.13. Ray Generation Shaders</td>
<td>499</td>
</tr>
<tr>
<td>9.15. Any-Hit Shaders</td>
<td>499</td>
</tr>
<tr>
<td>9.16. Closest Hit Shaders</td>
<td>500</td>
</tr>
<tr>
<td>9.17. Miss Shaders</td>
<td>500</td>
</tr>
<tr>
<td>9.18. Callable Shaders</td>
<td>500</td>
</tr>
<tr>
<td>9.19. Interpolation Decorations</td>
<td>500</td>
</tr>
<tr>
<td>9.20. Static Use</td>
<td>502</td>
</tr>
<tr>
<td>9.21. Scope</td>
<td>502</td>
</tr>
<tr>
<td>9.22. Group Operations</td>
<td>506</td>
</tr>
<tr>
<td>9.23. Quad Group Operations</td>
<td>508</td>
</tr>
<tr>
<td>9.24. Derivative Operations</td>
<td>508</td>
</tr>
<tr>
<td>9.25. Helper Invocations</td>
<td>510</td>
</tr>
<tr>
<td>9.27. Validation Cache</td>
<td>514</td>
</tr>
<tr>
<td>10. Pipelines</td>
<td>522</td>
</tr>
<tr>
<td>10.1. Compute Pipelines</td>
<td>523</td>
</tr>
<tr>
<td>10.2. Graphics Pipelines</td>
<td>537</td>
</tr>
<tr>
<td>10.3. Ray Tracing Pipelines</td>
<td>567</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>10.4. Pipeline Destruction</td>
<td>595</td>
</tr>
<tr>
<td>10.5. Multiple Pipeline Creation</td>
<td>596</td>
</tr>
<tr>
<td>10.6. Pipeline Derivatives</td>
<td>596</td>
</tr>
<tr>
<td>10.7. Pipeline Cache</td>
<td>597</td>
</tr>
<tr>
<td>10.8. Specialization Constants</td>
<td>605</td>
</tr>
<tr>
<td>10.9. Pipeline Libraries</td>
<td>609</td>
</tr>
<tr>
<td>10.10. Pipeline Binding</td>
<td>610</td>
</tr>
<tr>
<td>10.11. Dynamic State</td>
<td>616</td>
</tr>
<tr>
<td>10.12. Pipeline Shader Information</td>
<td>616</td>
</tr>
<tr>
<td>10.13. Pipeline Compiler Control</td>
<td>630</td>
</tr>
<tr>
<td>10.14. Pipeline Creation Feedback</td>
<td>631</td>
</tr>
<tr>
<td>11. Memory Allocation</td>
<td>635</td>
</tr>
<tr>
<td>11.1. Host Memory</td>
<td>635</td>
</tr>
<tr>
<td>11.2. Device Memory</td>
<td>642</td>
</tr>
<tr>
<td>12. Resource Creation</td>
<td>722</td>
</tr>
<tr>
<td>12.1. Buffers</td>
<td>722</td>
</tr>
<tr>
<td>12.2. Buffer Views</td>
<td>735</td>
</tr>
<tr>
<td>12.3. Images</td>
<td>740</td>
</tr>
<tr>
<td>12.4. Image Layouts</td>
<td>781</td>
</tr>
<tr>
<td>12.5. Image Views</td>
<td>787</td>
</tr>
<tr>
<td>12.6. Acceleration Structures</td>
<td>811</td>
</tr>
<tr>
<td>12.7. Resource Memory Association</td>
<td>841</td>
</tr>
<tr>
<td>12.8. Resource Sharing Mode</td>
<td>877</td>
</tr>
<tr>
<td>12.9. Memory Aliasing</td>
<td>879</td>
</tr>
<tr>
<td>13. Samplers</td>
<td>882</td>
</tr>
<tr>
<td>13.1. Sampler Y′C_bC_r conversion</td>
<td>894</td>
</tr>
<tr>
<td>14. Resource Descriptors</td>
<td>905</td>
</tr>
<tr>
<td>14.1. Descriptor Types</td>
<td>905</td>
</tr>
<tr>
<td>14.2. Descriptor Sets</td>
<td>909</td>
</tr>
<tr>
<td>14.3. Physical Storage Buffer Access</td>
<td>1001</td>
</tr>
<tr>
<td>15. Shader Interfaces</td>
<td>1005</td>
</tr>
<tr>
<td>15.1. Shader Input and Output Interfaces</td>
<td>1005</td>
</tr>
<tr>
<td>15.2. Vertex Input Interface</td>
<td>1009</td>
</tr>
<tr>
<td>15.3. Fragment Output Interface</td>
<td>1009</td>
</tr>
<tr>
<td>15.4. Fragment Input Attachment Interface</td>
<td>1010</td>
</tr>
<tr>
<td>15.5. Ray Tracing Pipeline Interface</td>
<td>1011</td>
</tr>
<tr>
<td>15.6. Shader Resource Interface</td>
<td>1012</td>
</tr>
<tr>
<td>15.7. Built-In Variables</td>
<td>1020</td>
</tr>
<tr>
<td>16. Image Operations</td>
<td>1070</td>
</tr>
<tr>
<td>16.1. Image Operations Overview</td>
<td>1070</td>
</tr>
<tr>
<td>16.2. Conversion Formulas</td>
<td>1074</td>
</tr>
</tbody>
</table>
16.3. Texel Input Operations ................................................. 1076
16.4. Texel Output Operations ........................................... 1093
16.5. Normalized Texel Coordinate Operations ......................... 1095
16.6. Unnormalized Texel Coordinate Operations ....................... 1102
16.7. Integer Texel Coordinate Operations .............................. 1104
16.8. Image Sample Operations ......................................... 1104
16.9. Texel Footprint Evaluation ....................................... 1108
16.10. Image Operation Steps ........................................... 1111
16.11. Image Query Instructions ........................................ 1111
17. Fragment Density Map Operations .................................. 1113
  17.1. Fragment Density Map Operations Overview ...................... 1113
  17.2. Fetch Density Value ........................................... 1113
  17.3. Fragment Area Conversion ..................................... 1114
18. Queries ................................................................. 1116
  18.1. Query Pools ...................................................... 1116
  18.2. Query Operation ................................................ 1123
  18.3. Occlusion Queries ............................................. 1149
  18.4. Pipeline Statistics Queries .................................... 1149
  18.5. Timestamp Queries ............................................ 1152
  18.6. Performance Queries .......................................... 1158
  18.7. Transform Feedback Queries .................................. 1162
  18.8. Intel performance queries .................................... 1162
  18.9. Result Status Queries ........................................ 1177
  18.10. Video Encode Bitstream Buffer Range ......................... 1177
19. Clear Commands ....................................................... 1178
  19.1. Clearing Images Outside A Render Pass Instance ............... 1178
  19.2. Clearing Images Inside A Render Pass Instance ............... 1185
  19.3. Clear Values .................................................... 1189
  19.4. Filling Buffers ................................................. 1190
  19.5. Updating Buffers .............................................. 1193
20. Copy Commands ......................................................... 1197
  20.1. Common Operation ............................................. 1197
  20.2. Copying Data Between Buffers ................................ 1198
  20.3. Copying Data Between Images .................................. 1205
  20.4. Copying Data Between Buffers and Images ..................... 1223
  20.5. Image Copies with Scaling .................................... 1254
  20.6. Resolving Multisample Images .................................. 1274
  20.7. Buffer Markers .................................................. 1280
21. Drawing Commands ...................................................... 1292
  21.1. Primitive Topologies ......................................... 1295
  21.2. Primitive Order ............................................... 1305
32.4. Sparse Partially-Resident Images .................................................. 1669
32.5. Sparse Memory Aliasing .............................................................. 1677
32.7. Sparse Resource API ............................................................... 1680
33. Window System Integration (WSI) .................................................. 1710
   33.1. WSI Platform ................................................................. 1710
   33.2. WSI Surface ................................................................. 1710
   33.3. Presenting Directly to Display Devices ................................. 1741
   33.4. Querying for WSI Support ................................................. 1773
   33.5. Surface Queries ............................................................ 1779
   33.6. Full Screen Exclusive Control ........................................... 1805
   33.7. Device Group Queries ..................................................... 1807
   33.8. Display Timing Queries .................................................... 1813
   33.9. WSI Swapchain ............................................................. 1819
   33.10. Hdr Metadata ............................................................... 1865
34. Deferred Host Operations .......................................................... 1868
   34.1. Requesting Deferral ........................................................ 1868
   34.2. Deferred Host Operations API ........................................... 1869
35. Private Data ........................................................................... 1875
36. Acceleration Structures ............................................................ 1881
   36.1. Acceleration Structures .................................................... 1881
   36.2. Host Acceleration Structure Operations ............................... 1937
37. Ray Traversal ......................................................................... 1954
   37.1. Ray Intersection Candidate Determination .............................. 1954
   37.2. Ray Intersection Culling .................................................. 1956
   37.3. Ray Intersection Confirmation .......................................... 1959
   37.4. Ray Closest Hit Determination .......................................... 1960
   37.5. Ray Result Determination ................................................ 1960
38. Ray Tracing ........................................................................... 1962
   38.1. Shader Call Instructions ................................................... 1962
   38.2. Ray Tracing Commands .................................................... 1964
   38.3. Shader Binding Table ....................................................... 1990
   38.4. Ray Tracing Pipeline Stack .............................................. 1993
39. Video Decode and Encode Operations ........................................ 1995
   39.2. Introduction .................................................................. 1996
   39.3. Video Physical Device Capabilities .................................... 2004
   39.4. Video Session Objects ...................................................... 2015
   39.5. Video Decode Operations ................................................ 2037
   39.6. Video Decode of AVC (ITU-T H.264) ................................... 2045
   39.7. Video Decode of HEVC (ITU-T H.265) ................................. 2053
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window System-Specific Header Control (Informative)</td>
<td>3103</td>
</tr>
<tr>
<td>Provisional Extension Header Control (Informative)</td>
<td>3105</td>
</tr>
<tr>
<td>Appendix G: Invariance</td>
<td>3106</td>
</tr>
<tr>
<td>Repeatability</td>
<td>3106</td>
</tr>
<tr>
<td>Multi-pass Algorithms</td>
<td>3106</td>
</tr>
<tr>
<td>Invariance Rules</td>
<td>3106</td>
</tr>
<tr>
<td>Tessellation Invariance</td>
<td>3108</td>
</tr>
<tr>
<td>Appendix H: Lexicon</td>
<td>3110</td>
</tr>
<tr>
<td>Glossary</td>
<td>3110</td>
</tr>
<tr>
<td>Common Abbreviations</td>
<td>3135</td>
</tr>
<tr>
<td>Prefixes</td>
<td>3136</td>
</tr>
<tr>
<td>Appendix I: Credits (Informative)</td>
<td>3138</td>
</tr>
<tr>
<td>Working Group Contributors to Vulkan</td>
<td>3138</td>
</tr>
<tr>
<td>Other Credits</td>
<td>3144</td>
</tr>
</tbody>
</table>
Chapter 1. Preamble

Copyright 2014-2021 The Khronos Group Inc.

This Specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos. Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this Specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

This Specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at https://www.khronos.org/files/member_agreement.pdf, and which defines the terms 'Scope', 'Compliant Portion', and 'Necessary Patent Claims'. Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos Intellectual Property Rights Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see https://www.khronos.org/adopters.

This Specification contains substantially unmodified functionality from, and is a successor to, Khronos specifications including OpenGL, OpenGL ES and OpenCL.

Some parts of this Specification are purely informative and so are EXCLUDED from the Scope of this Specification. The Document Conventions section of the Introduction defines how these parts of the Specification are identified.

Where this Specification uses technical terminology, defined in the Glossary or otherwise, that refer to enabling technologies that are not expressly set forth in this Specification, those enabling technologies are EXCLUDED from the Scope of this Specification. For clarity, enabling technologies not disclosed with particularity in this Specification (e.g. semiconductor manufacturing technology, hardware architecture, processor architecture or microarchitecture, memory architecture, compiler technology, object oriented technology, basic operating system technology, compression technology, algorithms, and so on) are NOT to be considered expressly set forth; only those application program interfaces and data structures disclosed with particularity are included in the Scope of this Specification.

For purposes of the Khronos Intellectual Property Rights Policy as it relates to the definition of Necessary Patent Claims, all recommended or optional features, behaviors and functionality set
forth in this Specification, if implemented, are considered to be included as Compliant Portions.

Where this Specification includes normative references to external documents, only the specifically identified sections of those external documents are INCLUDED in the Scope of this Specification. If not created by Khronos, those external documents may contain contributions from non-members of Khronos not covered by the Khronos Intellectual Property Rights Policy.

This document contains extensions which are not ratified by Khronos, and as such is not a ratified Specification, though it contains text from (and is a superset of) the ratified Vulkan Specification. The ratified versions of the Vulkan Specification can be found at https://www.khronos.org/registry/vulkan/specs/1.2/html/vkspec.html (core only) and https://www.khronos.org/registry/vulkan/specs/1.2-khr-extensions/html/vkspec.html (core with KHR extensions).

Vulkan and Khronos are registered trademarks of The Khronos Group Inc. ASTC is a trademark of ARM Holdings PLC; OpenCL is a trademark of Apple Inc.; and OpenGL and OpenGL ES are registered trademarks of Hewlett Packard Enterprise, all used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.
Chapter 2. Introduction

This document, referred to as the “Vulkan Specification” or just the “Specification” hereafter, describes the Vulkan Application Programming Interface (API). Vulkan is a C99 API designed for explicit control of low-level graphics and compute functionality.

The canonical version of the Specification is available in the official Vulkan Registry (https://www.khronos.org/registry/vulkan/). The source files used to generate the Vulkan specification are stored in the Vulkan Documentation Repository (https://github.com/KhronosGroup/Vulkan-Docs). The source repository additionally has a public issue tracker and allows the submission of pull requests that improve the specification.

2.1. Document Conventions

The Vulkan specification is intended for use by both implementors of the API and application developers seeking to make use of the API, forming a contract between these parties. Specification text may address either party; typically the intended audience can be inferred from context, though some sections are defined to address only one of these parties. (For example, Valid Usage sections only address application developers). Any requirements, prohibitions, recommendations or options defined by normative terminology are imposed only on the audience of that text.

Note

Structure and enumerated types defined in extensions that were promoted to core in a later version of Vulkan are now defined in terms of the equivalent Vulkan core interfaces. This affects the Vulkan Specification, the Vulkan header files, and the corresponding XML Registry.

2.1.1. Informative Language

Some language in the specification is purely informative, intended to give background or suggestions to implementors or developers.

If an entire chapter or section contains only informative language, its title will be suffixed with “(Informative)”.

All NOTEs are implicitly informative.

2.1.2. Normative Terminology

Within this specification, the key words must, required, should, recommended, may, and optional are to be interpreted as described in RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels (https://www.ietf.org/rfc/rfc2119.txt). These key words are highlighted in the specification for clarity. In text addressing application developers, their use expresses requirements that apply to application behavior. In text addressing implementors, their use expresses requirements that apply to implementations.

In text addressing application developers, the additional key words can and cannot are to be
interpreted as describing the capabilities of an application, as follows:

**can**
This word means that the application is able to perform the action described.

**cannot**
This word means that the API and/or the execution environment provide no mechanism through which the application can express or accomplish the action described.

These key words are never used in text addressing implementors.

*Note*
There is an important distinction between **cannot** and **must not**, as used in this Specification. **Cannot** means something the application literally is unable to express or accomplish through the API, while **must not** means something that the application is capable of expressing through the API, but that the consequences of doing so are undefined and potentially unrecoverable for the implementation (see Errors).

Unless otherwise noted in the section heading, all sections and appendices in this document are normative.

### 2.1.3. Technical Terminology

The Vulkan Specification makes use of common engineering and graphics terms such as **Pipeline**, **Shader**, and **Host** to identify and describe Vulkan API constructs and their attributes, states, and behaviors. The **Glossary** defines the basic meanings of these terms in the context of the Specification. The Specification text provides fuller definitions of the terms and may elaborate, extend, or clarify the **Glossary** definitions. When a term defined in the **Glossary** is used in normative language within the Specification, the definitions within the Specification govern and supersede any meanings the terms may have in other technical contexts (i.e. outside the Specification).

### 2.1.4. Normative References

References to external documents are considered normative references if the Specification uses any of the normative terms defined in **Normative Terminology** to refer to them or their requirements, either as a whole or in part.

The following documents are referenced by normative sections of the specification:


Chapter 3. Fundamentals

This chapter introduces fundamental concepts including the Vulkan architecture and execution model, API syntax, queues, pipeline configurations, numeric representation, state and state queries, and the different types of objects and shaders. It provides a framework for interpreting more specific descriptions of commands and behavior in the remainder of the Specification.

3.1. Host and Device Environment

The Vulkan Specification assumes and requires: the following properties of the host environment with respect to Vulkan implementations:

- The host must have runtime support for 8, 16, 32 and 64-bit signed and unsigned two's-complement integers, all addressable at the granularity of their size in bytes.
- The host must have runtime support for 32- and 64-bit floating-point types satisfying the range and precision constraints in the Floating Point Computation section.
- The representation and endianness of these types on the host must match the representation and endianness of the same types on every physical device supported.

Note

Since a variety of data types and structures in Vulkan may be accessible by both host and physical device operations, the implementation should be able to access such data efficiently in both paths in order to facilitate writing portable and performant applications.

3.2. Execution Model

This section outlines the execution model of a Vulkan system.

Vulkan exposes one or more devices, each of which exposes one or more queues which may process work asynchronously to one another. The set of queues supported by a device is partitioned into families. Each family supports one or more types of functionality and may contain multiple queues with similar characteristics. Queues within a single family are considered compatible with one another, and work produced for a family of queues can be executed on any queue within that family. This specification defines the following types of functionality that queues may support: video decode, video encode, graphics, compute, transfer and sparse memory management.

Note

A single device may report multiple similar queue families rather than, or as well as, reporting multiple members of one or more of those families. This indicates that while members of those families have similar capabilities, they are not directly compatible with one another.

Device memory is explicitly managed by the application. Each device may advertise one or more heaps, representing different areas of memory. Memory heaps are either device local or host local,
but are always visible to the device. Further detail about memory heaps is exposed via memory types available on that heap. Examples of memory areas that may be available on an implementation include:

- **device local** is memory that is physically connected to the device.
- **device local, host visible** is device local memory that is visible to the host.
- **host local, host visible** is memory that is local to the host and visible to the device and host.

On other architectures, there may only be a single heap that can be used for any purpose.

A Vulkan application controls a set of devices through the submission of command buffers which have recorded device commands issued via Vulkan library calls. The content of command buffers is specific to the underlying implementation and is opaque to the application. Once constructed, a command buffer can be submitted once or many times to a queue for execution. Multiple command buffers can be built in parallel by employing multiple threads within the application.

Command buffers submitted to different queues may execute in parallel or even out of order with respect to one another. Command buffers submitted to a single queue respect submission order, as described further in synchronization chapter. Command buffer execution by the device is also asynchronous to host execution. Once a command buffer is submitted to a queue, control may return to the application immediately. Synchronization between the device and host, and between different queues is the responsibility of the application.

### 3.2.1. Queue Operation

Vulkan queues provide an interface to the execution engines of a device. Commands for these execution engines are recorded into command buffers ahead of execution time. These command buffers are then submitted to queues with a queue submission command for execution in a number of batches. Once submitted to a queue, these commands will begin and complete execution without further application intervention, though the order of this execution is dependent on a number of implicit and explicit ordering constraints.

Work is submitted to queues using queue submission commands that typically take the form `vkQueue*` (e.g. `vkQueueSubmit`, `vkQueueBindSparse`), and optionally take a list of semaphores upon which to wait before work begins and a list of semaphores to signal once work has completed. The work itself, as well as signaling and waiting on the semaphores are all queue operations.

Queue operations on different queues have no implicit ordering constraints, and may execute in any order. Explicit ordering constraints between queues can be expressed with semaphores and fences.

Command buffer submissions to a single queue respect submission order and other implicit ordering guarantees, but otherwise may overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g. sparse memory binding) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with semaphores and fences.

Before a fence or semaphore is signaled, it is guaranteed that any previously submitted queue operations have completed execution, and that memory writes from those queue operations are...
available to future queue operations. Waiting on a signaled semaphore or fence guarantees that previous writes that are available are also visible to subsequent commands.

Command buffer boundaries, both between primary command buffers of the same or different batches or submissions as well as between primary and secondary command buffers, do not introduce any additional ordering constraints. In other words, submitting the set of command buffers (which can include executing secondary command buffers) between any semaphore or fence operations execute the recorded commands as if they had all been recorded into a single primary command buffer, except that the current state is reset on each boundary. Explicit ordering constraints can be expressed with explicit synchronization primitives.

There are a few implicit ordering guarantees between commands within a command buffer, but only covering a subset of execution. Additional explicit ordering constraints can be expressed with the various explicit synchronization primitives.

Note
Implementations have significant freedom to overlap execution of work submitted to a queue, and this is common due to deep pipelining and parallelism in Vulkan devices.

Commands recorded in command buffers either perform actions (draw, dispatch, clear, copy, query/timestamp operations, begin/end subpass operations), set state (bind pipelines, descriptor sets, and buffers, set dynamic state, push constants, set render pass/subpass state), or perform synchronization (set/wait events, pipeline barrier, render pass/subpass dependencies). Some commands perform more than one of these tasks. State setting commands update the current state of the command buffer. Some commands that perform actions (e.g. draw/dispatch) do so based on the current state set cumulatively since the start of the command buffer. The work involved in performing action commands is often allowed to overlap or to be reordered, but doing so must not alter the state to be used by each action command. In general, action commands are those commands that alter framebuffer attachments, read/write buffer or image memory, or write to query pools.

Synchronization commands introduce explicit execution and memory dependencies between two sets of action commands, where the second set of commands depends on the first set of commands. These dependencies enforce that both the execution of certain pipeline stages in the later set occur after the execution of certain stages in the source set, and that the effects of memory accesses performed by certain pipeline stages occur in order and are visible to each other. When not enforced by an explicit dependency or implicit ordering guarantees, action commands may overlap execution or execute out of order, and may not see the side effects of each other’s memory accesses.

The device executes queue operations asynchronously with respect to the host. Control is returned to an application immediately following command buffer submission to a queue. The application must synchronize work between the host and device as needed.

3.3. Object Model

The devices, queues, and other entities in Vulkan are represented by Vulkan objects. At the API
level, all objects are referred to by handles. There are two classes of handles, dispatchable and non-
dispatchable. Dispatchable handle types are a pointer to an opaque type. This pointer may be used
by layers as part of intercepting API commands, and thus each API command takes a dispatchable
type as its first parameter. Each object of a dispatchable type must have a unique handle value
during its lifetime.

Non-dispatchable handle types are a 64-bit integer type whose meaning is implementation-
dependent. If the privateData feature is enabled for a VkDevice, each object of a non-dispatchable
type created on that device must have a handle value that is unique among objects created on that
device, for the duration of the object’s lifetime. Otherwise, non-dispatchable handles may encode
object information directly in the handle rather than acting as a reference to an underlying object,
and thus may not have unique handle values. If handle values are not unique, then destroying one
such handle must not cause identical handles of other types to become invalid, and must not cause
identical handles of the same type to become invalid if that handle value has been created more
times than it has been destroyed.

All objects created or allocated from a VkDevice (i.e. with a VkDevice as the first parameter) are
private to that device, and must not be used on other devices.

3.3.1. Object Lifetime

Objects are created or allocated by vkCreate* and vkAllocate* commands, respectively. Once an
object is created or allocated, its “structure” is considered to be immutable, though the contents of
certain object types is still free to change. Objects are destroyed or freed by vkDestroy* and vkFree*
commands, respectively.

Objects that are allocated (rather than created) take resources from an existing pool object or
memory heap, and when freed return resources to that pool or heap. While object creation and
destruction are generally expected to be low-frequency occurrences during runtime, allocating and
freeing objects can occur at high frequency. Pool objects help accommodate improved performance
of the allocations and frees.

It is an application’s responsibility to track the lifetime of Vulkan objects, and not to destroy them
while they are still in use.

The ownership of application-owned memory is immediately acquired by any Vulkan command it
is passed into. Ownership of such memory must be released back to the application at the end of
the duration of the command, so that the application can alter or free this memory as soon as all
the commands that acquired it have returned.

The following object types are consumed when they are passed into a Vulkan command and not
further accessed by the objects they are used to create. They must not be destroyed in the duration
of any API command they are passed into:

- VkShaderModule
- VkPipelineCache
- VkValidationCacheEXT

A VkRenderPass object passed as a parameter to create another object is not further accessed by that
object after the duration of the command it is passed into. A *VkRenderPass* used in a command buffer follows the rules described below.

A *VkPipelineLayout* object **must** not be destroyed while any command buffer that uses it is in the recording state.

*VkDescriptorSetLayout* objects **may** be accessed by commands that operate on descriptor sets allocated using that layout, and those descriptor sets **must** not be updated with *vkUpdateDescriptorSets* after the descriptor set layout has been destroyed. Otherwise, a *VkDescriptorSetLayout* object passed as a parameter to create another object is not further accessed by that object after the duration of the command it is passed into.

The application **must** not destroy any other type of Vulkan object until all uses of that object by the device (such as via command buffer execution) have completed.

The following Vulkan objects **must** not be destroyed while any command buffers using the object are in the **pending state**:

- *VkEvent*
- *VkQueryPool*
- *VkBuffer*
- *VkBufferView*
- *VkImage*
- *VkImageView*
- *VkPipeline*
- *VkSampler*
- *VkSamplerYcbcrConversion*
- *VkDescriptorPool*
- *VkFramebuffer*
- *VkRenderPass*
- *VkCommandBuffer*
- *VkCommandPool*
- *VkDeviceMemory*
- *VkDescriptorSet*
- *VkIndirectCommandsLayoutNV*
  - *VkAccelerationStructureNV*
  - *VkAccelerationStructureKHR*

Destroying these objects will move any command buffers that are in the **recording or executable state**, and are using those objects, to the **invalid state**.

The following Vulkan objects **must** not be destroyed while any queue is executing commands that
In general, objects can be destroyed or freed in any order, even if the object being freed is involved in the use of another object (e.g. use of a resource in a view, use of a view in a descriptor set, use of a pipeline library in another pipeline, use of a referenced pipeline for additional graphics shader groups in another pipeline, use of a bottom level acceleration structure in an instance referenced by a top level acceleration structure, use of an object in a command buffer, binding of a memory allocation to a resource), as long as any object that uses the freed object is not further used in any way except to be destroyed or to be reset in such a way that it no longer uses the other object (such as resetting a command buffer). If the object has been reset, then it can be used as if it never used the freed object. An exception to this is when there is a parent/child relationship between objects. In this case, the application must not destroy a parent object before its children, except when the parent is explicitly defined to free its children when it is destroyed (e.g. for pool objects, as defined below).

VkCommandPool objects are parents of VkCommandBuffer objects. VkDescriptorPool objects are parents of VkDescriptorSet objects. VkDevice objects are parents of many object types (all that take a VkDevice as a parameter to their creation).

The following Vulkan objects have specific restrictions for when they can be destroyed:

- VkQueue objects cannot be explicitly destroyed. Instead, they are implicitly destroyed when the VkDevice object they are retrieved from is destroyed.
- Destroying a pool object implicitly frees all objects allocated from that pool. Specifically, destroying VkCommandPool frees all VkCommandBuffer objects that were allocated from it, and destroying VkDescriptorPool frees all VkDescriptorSet objects that were allocated from it.
- VkDevice objects can be destroyed when all VkQueue objects retrieved from them are idle, and all objects created from them have been destroyed. This includes the following objects:
  - VkFence
  - VkSemaphore
  - VkEvent
  - VkQueryPool
  - VkBuffer
  - VkBufferView
  - VkImage
  - VkImageView
  - VkShaderModule
  - VkPipelineCache
• **VkPhysicalDevice** objects cannot be explicitly destroyed. Instead, they are implicitly destroyed when the **VkInstance** object they are retrieved from is destroyed.

• **VkInstance** objects can be destroyed once all **VkDevice** objects created from any of its **VkPhysicalDevice** objects have been destroyed.

### 3.3.2. External Object Handles

As defined above, the scope of object handles created or allocated from a **VkDevice** is limited to that logical device. Objects which are not in scope are said to be external. To bring an external object into scope, an external handle must be exported from the object in the source scope and imported into the destination scope.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The scope of external handles and their associated resources may vary according to their type, but they can generally be shared across process and API boundaries.</td>
</tr>
</tbody>
</table>

### 3.4. Application Binary Interface

The mechanism by which Vulkan is made available to applications is platform- or implementation-defined. On many platforms the C interface described in this Specification is provided by a shared library. Since shared libraries can be changed independently of the applications that use them, they present particular compatibility challenges, and this Specification places some requirements on them.

Shared library implementations must use the default Application Binary Interface (ABI) of the standard C compiler for the platform, or provide customized API headers that cause application code to use the implementation’s non-default ABI. An ABI in this context means the size, alignment,
and layout of C data types; the procedure calling convention; and the naming convention for shared library symbols corresponding to C functions. Customizing the calling convention for a platform is usually accomplished by defining calling convention macros appropriately in `vk_platform.h`.

On platforms where Vulkan is provided as a shared library, library symbols beginning with “vk” and followed by a digit or uppercase letter are reserved for use by the implementation. Applications which use Vulkan must not provide definitions of these symbols. This allows the Vulkan shared library to be updated with additional symbols for new API versions or extensions without causing symbol conflicts with existing applications.

Shared library implementations should provide library symbols for commands in the highest version of this Specification they support, and for Window System Integration extensions relevant to the platform. They may also provide library symbols for commands defined by additional extensions.

Note

These requirements and recommendations are intended to allow implementors to take advantage of platform-specific conventions for SDKs, ABIs, library versioning mechanisms, etc. while still minimizing the code changes necessary to port applications or libraries between platforms. Platform vendors, or providers of the de facto standard Vulkan shared library for a platform, are encouraged to document what symbols the shared library provides and how it will be versioned when new symbols are added.

Applications should only rely on shared library symbols for commands in the minimum core version required by the application. `vkGetInstanceProcAddr` and `vkGetDeviceProcAddr` should be used to obtain function pointers for commands in core versions beyond the application's minimum required version.

3.5. Command Syntax and Duration

The Specification describes Vulkan commands as functions or procedures using C99 syntax. Language bindings for other languages such as C++ and JavaScript may allow for stricter parameter passing, or object-oriented interfaces.

Vulkan uses the standard C types for the base type of scalar parameters (e.g. types from `<stdint.h>`), with exceptions described below, or elsewhere in the text when appropriate:

`VkBool32` represents boolean `True` and `False` values, since C does not have a sufficiently portable built-in boolean type:

```c
// Provided by VK_VERSION_1_0
typedef uint32_t VkBool32;
```

`VK_TRUE` represents a boolean `True` (unsigned integer 1) value, and `VK_FALSE` a boolean `False` (unsigned integer 0) value.
All values returned from a Vulkan implementation in a `VkBool32` will be either `VK_TRUE` or `VK_FALSE`.

Applications **must** not pass any other values than `VK_TRUE` or `VK_FALSE` into a Vulkan implementation where a `VkBool32` is expected.

`VkDeviceSize` represents device memory size and offset values:

```c
// Provided by VK_VERSION_1_0
typedef uint64_t VkDeviceSize;
```

`VkDeviceAddress` represents device buffer address values:

```c
// Provided by VK_VERSION_1_0
typedef uint64_t VkDeviceAddress;
```

Commands that create Vulkan objects are of the form `vkCreate*` and take `Vk*CreateInfo` structures with the parameters needed to create the object. These Vulkan objects are destroyed with commands of the form `vkDestroy*`.

The last in-parameter to each command that creates or destroys a Vulkan object is `pAllocator`. The `pAllocator` parameter *can* be set to a non-NULL value such that allocations for the given object are delegated to an application provided callback; refer to the Memory Allocation chapter for further details.

Commands that allocate Vulkan objects owned by pool objects are of the form `vkAllocate*`, and take `Vk*AllocateInfo` structures. These Vulkan objects are freed with commands of the form `vkFree*`. These objects do not take allocators; if host memory is needed, they will use the allocator that was specified when their parent pool was created.

Commands are recorded into a command buffer by calling API commands of the form `vkCmd*`. Each such command *may* have different restrictions on where it *can* be used: in a primary and/or secondary command buffer, inside and/or outside a render pass, and in one or more of the supported queue types. These restrictions are documented together with the definition of each such command.

The *duration* of a Vulkan command refers to the interval between calling the command and its return to the caller.

### 3.5.1. Lifetime of Retrieved Results

Information is retrieved from the implementation with commands of the form `vkGet*` and `vkEnumerate*`.

Unless otherwise specified for an individual command, the results are *invariant*; that is, they will remain unchanged when retrieved again by calling the same command with the same parameters, so long as those parameters themselves all remain valid.
3.6. Threading Behavior

Vulkan is intended to provide scalable performance when used on multiple host threads. All commands support being called concurrently from multiple threads, but certain parameters, or components of parameters are defined to be externally synchronized. This means that the caller must guarantee that no more than one thread is using such a parameter at a given time.

More precisely, Vulkan commands use simple stores to update the state of Vulkan objects. A parameter declared as externally synchronized may have its contents updated at any time during the host execution of the command. If two commands operate on the same object and at least one of the commands declares the object to be externally synchronized, then the caller must guarantee not only that the commands do not execute simultaneously, but also that the two commands are separated by an appropriate memory barrier (if needed).

Note

Memory barriers are particularly relevant for hosts based on the ARM CPU architecture, which is more weakly ordered than many developers are accustomed to from x86/x64 programming. Fortunately, most higher-level synchronization primitives (like the pthread library) perform memory barriers as a part of mutual exclusion, so muting Vulkan objects via these primitives will have the desired effect.

Similarly the application must avoid any potential data hazard of application-owned memory that has its ownership temporarily acquired by a Vulkan command. While the ownership of application-owned memory remains acquired by a command the implementation may read the memory at any point, and it may write non-const qualified memory at any point. Parameters referring to non-const qualified application-owned memory are not marked explicitly as externally synchronized in the Specification.

If an application is using deferred host operations in a command, and that operation is successfully deferred, object parameters and application-owned memory passed to that command may be accessed at any time until the deferred operation is complete.

Many object types are immutable, meaning the objects cannot change once they have been created. These types of objects never need external synchronization, except that they must not be destroyed while they are in use on another thread. In certain special cases mutable object parameters are internally synchronized, making external synchronization unnecessary. Any command parameters that are not labeled as externally synchronized are either not mutated by the command or are internally synchronized. Additionally, certain objects related to a command’s parameters (e.g. command pools and descriptor pools) may be affected by a command, and must also be externally synchronized. These implicit parameters are documented as described below.

Parameters of commands that are externally synchronized are listed below.
Externally Synchronized Parameters

- The `instance` parameter in `vkDestroyInstance`
- The `device` parameter in `vkDestroyDevice`
- The `queue` parameter in `vkQueueSubmit`
- The `fence` parameter in `vkQueueSubmit`
- The `queue` parameter in `vkQueueWaitIdle`
- The `memory` parameter in `vkFreeMemory`
- The `memory` parameter in `vkMapMemory`
- The `memory` parameter in `vkUnmapMemory`
- The `buffer` parameter in `vkBindBufferMemory`
- The `image` parameter in `vkBindImageMemory`
- The `queue` parameter in `vkQueueBindSparse`
- The `fence` parameter in `vkQueueBindSparse`
- The `fence` parameter in `vkDestroyFence`
- The `semaphore` parameter in `vkDestroySemaphore`
- The `event` parameter in `vkDestroyEvent`
- The `event` parameter in `vkSetEvent`
- The `event` parameter in `vkResetEvent`
- The `queryPool` parameter in `vkDestroyQueryPool`
- The `buffer` parameter in `vkDestroyBuffer`
- The `bufferView` parameter in `vkDestroyBufferView`
- The `image` parameter in `vkDestroyImage`
- The `imageView` parameter in `vkDestroyImageView`
- The `shaderModule` parameter in `vkDestroyShaderModule`
- The `pipelineCache` parameter in `vkDestroyPipelineCache`
- The `dstCache` parameter in `vkMergePipelineCaches`
- The `pipeline` parameter in `vkDestroyPipeline`
- The `pipelineLayout` parameter in `vkDestroyPipelineLayout`
- The `sampler` parameter in `vkDestroySampler`
- The `descriptorSetLayout` parameter in `vkDestroyDescriptorSetLayout`
- The `descriptorPool` parameter in `vkDestroyDescriptorPool`
- The `descriptorPool` parameter in `vkResetDescriptorPool`
- The `descriptorPool` member of the `pAllocateInfo` parameter in `vkAllocateDescriptorSets`
- The `descriptorPool` parameter in `vkFreeDescriptorSets`
• The `framebuffer` parameter in `vkDestroyFramebuffer`
• The `renderPass` parameter in `vkDestroyRenderPass`
• The `commandPool` parameter in `vkDestroyCommandPool`
• The `commandPool` parameter in `vkResetCommandPool`
• The `commandPool` member of the `pAllocateInfo` parameter in `vkAllocateCommandBuffers`
• The `commandPool` parameter in `vkFreeCommandBuffers`
• The `commandBuffer` parameter in `vkBeginCommandBuffer`
• The `commandBuffer` parameter in `vkEndCommandBuffer`
• The `commandBuffer` parameter in `vkResetCommandBuffer`
• The `commandBuffer` parameter in `vkCmdBindPipeline`
• The `commandBuffer` parameter in `vkCmdSetViewport`
• The `commandBuffer` parameter in `vkCmdSetScissor`
• The `commandBuffer` parameter in `vkCmdSetLineWidth`
• The `commandBuffer` parameter in `vkCmdSetDepthBias`
• The `commandBuffer` parameter in `vkCmdSetBlendConstants`
• The `commandBuffer` parameter in `vkCmdSetDepthBounds`
• The `commandBuffer` parameter in `vkCmdSetStencilCompareMask`
• The `commandBuffer` parameter in `vkCmdSetStencilWriteMask`
• The `commandBuffer` parameter in `vkCmdSetStencilReference`
• The `commandBuffer` parameter in `vkCmdBindDescriptorSets`
• The `commandBuffer` parameter in `vkCmdBindIndexBuffer`
• The `commandBuffer` parameter in `vkCmdBindVertexBuffers`
• The `commandBuffer` parameter in `vkCmdDraw`
• The `commandBuffer` parameter in `vkCmdDrawIndexed`
• The `commandBuffer` parameter in `vkCmdDrawIndirect`
• The `commandBuffer` parameter in `vkCmdDispatch`
• The `commandBuffer` parameter in `vkCmdDispatchIndirect`
• The `commandBuffer` parameter in `vkCmdCopyBuffer`
• The `commandBuffer` parameter in `vkCmdCopyImage`
• The `commandBuffer` parameter in `vkCmdBlitImage`
• The `commandBuffer` parameter in `vkCmdCopyBufferToImage`
• The `commandBuffer` parameter in `vkCmdCopyImageToBuffer`
• The `commandBuffer` parameter in `vkCmdUpdateBuffer`
• The `commandBuffer` parameter in `vkCmdFillBuffer`
• The `commandBuffer` parameter in `vkCmdClearColorImage`
• The `commandBuffer` parameter in `vkCmdClearDepthStencilImage`
• The `commandBuffer` parameter in `vkCmdClearAttachments`
• The `commandBuffer` parameter in `vkCmdResolveImage`
• The `commandBuffer` parameter in `vkCmdSetEvent`
• The `commandBuffer` parameter in `vkCmdResetEvent`
• The `commandBuffer` parameter in `vkCmdWaitEvents`
• The `commandBuffer` parameter in `vkCmdPipelineBarrier`
• The `commandBuffer` parameter in `vkCmdBeginQuery`
• The `commandBuffer` parameter in `vkCmdEndQuery`
• The `commandBuffer` parameter in `vkCmdResetQueryPool`
• The `commandBuffer` parameter in `vkCmdWriteTimestamp`
• The `commandBuffer` parameter in `vkCmdCopyQueryPoolResults`
• The `commandBuffer` parameter in `vkCmdPushConstants`
• The `commandBuffer` parameter in `vkCmdBeginRenderPass`
• The `commandBuffer` parameter in `vkCmdNextSubpass`
• The `commandBuffer` parameter in `vkCmdEndRenderPass`
• The `commandBuffer` parameter in `vkCmdExecuteCommands`
• The `commandBuffer` parameter in `vkCmdSetDeviceMask`
• The `commandPool` parameter in `vkTrimDispatchBase`
• The `ycbcrConversion` parameter in `vkDestroySamplerYcbcrConversion`
• The `descriptorUpdateTemplate` parameter in `vkDestroyDescriptorUpdateTemplate`
• The `descriptorSet` parameter in `vkUpdateDescriptorSetWithTemplate`
• The `surface` parameter in `vkDestroySurfaceKHR`
• The `surface` member of the `pCreateInfo` parameter in `vkCreateSwapchainKHR`
• The `oldSwapchain` member of the `pCreateInfo` parameter in `vkCreateSwapchainKHR`
• The `swapchain` parameter in `vkDestroySwapchainKHR`
• The `swapchain` parameter in `vkAcquireNextImageKHR`
• The `semaphore` parameter in `vkAcquireNextImageKHR`
• The `fence` parameter in `vkAcquireNextImageKHR`
• The `queue` parameter in `vkQueuePresentKHR`
• The `surface` parameter in `vkGetDeviceGroupSurfacePresentModesKHR`
• The `surface` parameter in `vkGetPhysicalDevicePresentRectanglesKHR`
• The `display` parameter in `vkCreateDisplayModeKHR`
• The `mode` parameter in `vkGetDisplayPlaneCapabilitiesKHR`
• The `commandBuffer` parameter in `vkCmdSetDeviceMaskKHR`
• The `commandBuffer` parameter in `vkCmdDispatchBaseKHR`
• The `commandPool` parameter in `vkTrimCommandPoolKHR`
• The `commandBuffer` parameter in `vkCmdPushDescriptorSetKHR`
• The `commandBuffer` parameter in `vkCmdPushDescriptorSetWithTemplateKHR`
• The `descriptorUpdateTemplate` parameter in `vkDestroyDescriptorUpdateTemplateKHR`
• The `descriptorSet` parameter in `vkUpdateDescriptorSetWithTemplateKHR`
• The `commandBuffer` parameter in `vkCmdBeginRenderPass2`
• The `commandBuffer` parameter in `vkCmdBeginRenderPass2KHR`
• The `commandBuffer` parameter in `vkCmdNextSubpass2`
• The `commandBuffer` parameter in `vkCmdNextSubpass2KHR`
• The `commandBuffer` parameter in `vkCmdEndRenderPass2`
• The `commandBuffer` parameter in `vkCmdEndRenderPass2KHR`
• The `swapchain` parameter in `vkGetSwapchainStatusKHR`
• The `ycbcrConversion` parameter in `vkDestroySamplerYcbcrConversionKHR`
• The `commandBuffer` parameter in `vkCmdDrawIndirectCount`
• The `commandBuffer` parameter in `vkCmdDrawIndirectCountKHR`
• The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCount`
• The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCountKHR`
• The `operation` parameter in `vkDestroyDeferredOperationKHR`
• The `commandBuffer` parameter in `vkCmdSetEvent2KHR`
• The `commandBuffer` parameter in `vkCmdResetEvent2KHR`
• The `commandBuffer` parameter in `vkCmdWaitEvents2KHR`
• The `commandBuffer` parameter in `vkCmdPipelineBarrier2KHR`
• The `commandBuffer` parameter in `vkCmdWriteTimestamp2KHR`
• The `queue` parameter in `vkQueueSubmit2KHR`
• The `fence` parameter in `vkQueueSubmit2KHR`
• The `commandBuffer` parameter in `vkCmdWriteBufferMarker2AMD`
• The `commandBuffer` parameter in `vkCmdCopyBuffer2KHR`
• The `commandBuffer` parameter in `vkCmdCopyImage2KHR`
• The `commandBuffer` parameter in `vkCmdCopyBufferToImage2KHR`
• The `commandBuffer` parameter in `vkCmdCopyImageToBuffer2KHR`
• The `commandBuffer` parameter in `vkCmdBlitImage2KHR`
• The `commandBuffer` parameter in `vkCmdResolveImage2KHR`
• The `callback` parameter in `vkDestroyDebugReportCallbackEXT`
• The `object` member of the `pTagInfo` parameter in `vkDebugMarkerSetObjectTagEXT`
• The `object` member of the `pNameInfo` parameter in `vkDebugMarkerSetObjectNameEXT`
• The `commandBuffer` parameter in `vkCmdDebugMarkerBeginEXT`
• The `commandBuffer` parameter in `vkCmdDebugMarkerEndEXT`
• The `commandBuffer` parameter in `vkCmdDebugMarkerInsertEXT`
• The `commandBuffer` parameter in `vkCmdBindTransformFeedbackBuffersEXT`
• The `commandBuffer` parameter in `vkCmdBeginTransformFeedbackEXT`
• The `commandBuffer` parameter in `vkCmdEndTransformFeedbackEXT`
• The `commandBuffer` parameter in `vkCmdBeginQueryIndexedEXT`
• The `commandBuffer` parameter in `vkCmdEndQueryIndexedEXT`
• The `commandBuffer` parameter in `vkCmdDrawIndirectByteCountEXT`
• The `commandBuffer` parameter in `vkCmdDrawIndirectCountAMD`
• The `commandBuffer` parameter in `vkCmdDrawIndexedIndirectCountAMD`
• The `commandBuffer` parameter in `vkCmdBeginConditionalRenderingEXT`
• The `commandBuffer` parameter in `vkCmdEndConditionalRenderingEXT`
• The `commandBuffer` parameter in `vkCmdSetViewportWScalingNV`
• The `swapchain` parameter in `vkGetRefreshCycleDurationGOOGLE`
• The `swapchain` parameter in `vkGetPastPresentationTimingGOOGLE`
• The `commandBuffer` parameter in `vkCmdSetDiscardRectangleEXT`
• The `objectHandle` member of the `pNameInfo` parameter in `vkSetDebugUtilsObjectNameEXT`
• The `objectHandle` member of the `pTagInfo` parameter in `vkSetDebugUtilsObjectTagEXT`
• The `commandBuffer` parameter in `vkCmdBeginDebugUtilsLabelEXT`
• The `commandBuffer` parameter in `vkCmdEndDebugUtilsLabelEXT`
• The `commandBuffer` parameter in `vkCmdInsertDebugUtilsLabelEXT`
• The `messenger` parameter in `vkDestroyDebugUtilsMessengerEXT`
• The `commandBuffer` parameter in `vkCmdSetSampleLocationsEXT`
• The `validationCache` parameter in `vkDestroyValidationCacheEXT`
• The `dstCache` parameter in `vkMergeValidationCachesEXT`
• The `commandBuffer` parameter in `vkCmdBindShadingRateImageNV`
• The `commandBuffer` parameter in `vkCmdSetViewportShadingRatePaletteNV`
• The `commandBuffer` parameter in `vkCmdSetCoarseSampleOrderNV`
• The `accelerationStructure` parameter in `vkDestroyAccelerationStructureNV`
• The `commandBuffer` parameter in `vkCmdBuildAccelerationStructureNV`
• The `commandBuffer` parameter in `vkCmdCopyAccelerationStructureNV`
• The `commandBuffer` parameter in `vkCmdTraceRaysNV`
• The `commandBuffer` parameter in `vkCmdWriteAccelerationStructuresPropertiesNV`
• The `commandBuffer` parameter in `vkCmdWriteBufferMarkerAMD`
• The `commandBuffer` parameter in `vkCmdDrawMeshTasksNV`
• The `commandBuffer` parameter in `vkCmdDrawMeshTasksIndirectNV`
• The `commandBuffer` parameter in `vkCmdDrawMeshTasksIndirectCountNV`
• The `commandBuffer` parameter in `vkCmdSetExclusiveScissorNV`
• The `commandBuffer` parameter in `vkCmdSetCheckpointNV`
• The `commandBuffer` parameter in `vkCmdSetPerformanceMarkerINTEL`
• The `commandBuffer` parameter in `vkCmdSetPerformanceStreamMarkerINTEL`
• The `commandBuffer` parameter in `vkCmdSetPerformanceOverrideINTEL`
• The `configuration` parameter in `vkReleasePerformanceConfigurationINTEL`
• The `commandBuffer` parameter in `vkCmdSetLineStippleEXT`
• The `commandBuffer` parameter in `vkCmdSetCullModeEXT`
• The `commandBuffer` parameter in `vkCmdSetFrontFaceEXT`
• The `commandBuffer` parameter in `vkCmdSetPrimitiveTopologyEXT`
• The `commandBuffer` parameter in `vkCmdSetViewportWithCountEXT`
• The `commandBuffer` parameter in `vkCmdSetScissorWithCountEXT`
• The `commandBuffer` parameter in `vkCmdBindVertexBuffers2EXT`
• The `commandBuffer` parameter in `vkCmdSetDepthTestEnableEXT`
• The `commandBuffer` parameter in `vkCmdSetDepthWriteEnableEXT`
• The `commandBuffer` parameter in `vkCmdSetDepthCompareOpEXT`
• The `commandBuffer` parameter in `vkCmdSetDepthBoundsTestEnableEXT`
• The `commandBuffer` parameter in `vkCmdSetStencilTestEnableEXT`
• The `commandBuffer` parameter in `vkCmdSetStencilOpEXT`
• The `commandBuffer` parameter in `vkCmdPreprocessGeneratedCommandsNV`
• The `commandBuffer` parameter in `vkCmdExecuteGeneratedCommandsNV`
• The `commandBuffer` parameter in `vkCmdBindPipelineShaderGroupNV`
• The `indirectCommandsLayout` parameter in `vkDestroyIndirectCommandsLayoutNV`
• The `privateDataSlot` parameter in `vkDestroyPrivateDataSlotEXT`
• The `commandBuffer` parameter in `vkCmdSetFragmentShadingRateEnumNV`
• The `commandBuffer` parameter in `vkCmdSetVertexInputEXT`
• The `commandBuffer` parameter in `vkCmdSetPatchControlPointsEXT`
• The `commandBuffer` parameter in `vkCmdSetRasterizerDiscardEnableEXT`
• The `commandBuffer` parameter in `vkCmdSetDepthBiasEnableEXT`  
• The `commandBuffer` parameter in `vkCmdSetLogicOpEXT`  
• The `commandBuffer` parameter in `vkCmdSetPrimitiveRestartEnableEXT`  
• The `commandBuffer` parameter in `vkCmdSetColorWriteEnableEXT`  
• The `accelerationStructure` parameter in `vkDestroyAccelerationStructureKHR`  
• The `commandBuffer` parameter in `vkCmdBuildAccelerationStructuresKHR`  
• The `commandBuffer` parameter in `vkCmdBuildAccelerationStructuresIndirectKHR`  
• The `commandBuffer` parameter in `vkCmdCopyAccelerationStructureToMemoryKHR`  
• The `commandBuffer` parameter in `vkCmdCopyMemoryToAccelerationStructureKHR`  
• The `commandBuffer` parameter in `vkCmdWriteAccelerationStructuresPropertiesKHR`  
• The `commandBuffer` parameter in `vkCmdTraceRaysKHR`  
• The `commandBuffer` parameter in `vkCmdTraceRaysIndirectKHR`  
• The `commandBuffer` parameter in `vkCmdSetRayTracingPipelineStackSizeKHR`  

For `VkPipelineCache` objects created with `flags` containing `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT`, the above table is extended with the `pipelineCache` parameter to `vkCreate*Pipelines` being externally synchronized.

There are also a few instances where a command can take in a user allocated list whose contents are externally synchronized parameters. In these cases, the caller must guarantee that at most one thread is using a given element within the list at a given time. These parameters are listed below.
Externally Synchronized Parameter Lists

- The buffer member of each element of the pBufferBinds member of each element of the pBindInfo parameter in vkQueueBindSparse
- The image member of each element of the pImageOpaqueBinds member of each element of the pBindInfo parameter in vkQueueBindSparse
- The image member of each element of the pImageBinds member of each element of the pBindInfo parameter in vkQueueBindSparse
- Each element of the pFences parameter in vkResetFences
- Each element of the pDescriptorSets parameter in vkFreeDescriptorSets
- The dstSet member of each element of the pDescriptorWrites parameter in vkUpdateDescriptorSets
- The dstSet member of each element of the pDescriptorCopies parameter in vkUpdateDescriptorSets
- Each element of the pCommandBuffers parameter in vkFreeCommandBuffers
- Each element of the pWaitSemaphores member of the pPresentInfo parameter in vkQueuePresentKHR
- Each element of the pSwapchains member of the pPresentInfo parameter in vkQueuePresentKHR
- The surface member of each element of the pCreateInfos parameter in vkCreateSharedSwapchainsKHR
- The oldSwapchain member of each element of the pCreateInfos parameter in vkCreateSharedSwapchainsKHR

In addition, there are some implicit parameters that need to be externally synchronized. For example, all commandBuffer parameters that need to be externally synchronized imply that the commandPool that was passed in when creating that command buffer also needs to be externally synchronized. The implicit parameters and their associated object are listed below.
Implicit Externally Synchronized Parameters

- All `VkPhysicalDevice` objects enumerated from `instance` in `vkDestroyInstance`.
- All `VkQueue` objects received from `device` in `vkDestroyDevice`.
- All `VkQueue` objects created from `device` in `vkDeviceWaitIdle`.
- Any `VkDescriptorSet` objects allocated from `descriptorPool` in `vkResetDescriptorPool`.
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkBeginCommandBuffer`.
- The `VkCommandPool` that `commandBuffer` was allocated from in `vkEndCommandBuffer`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindPipeline`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetViewport`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetScissor`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetLineWidth`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDepthBias`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetBlendConstants`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetDepthBounds`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilCompareMask`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetStencilWriteMask`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindDescriptorSets`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindIndexBuffer`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindVertexBuffers`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDraw`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexed`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirect`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDispatch`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDispatchIndirect`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyBuffer`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyImage`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBlitImage`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyBufferToImage`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyImageToBuffer`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdUpdateBuffer`.
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdFillBuffer`.
• The VkCommandPool that commandBuffer was allocated from, in vkCmdClearColorImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdClearDepthStencilImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdClearAttachments
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResolveImage
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetEvent
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResetEvent
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPipelineBarrier
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginQuery
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndQuery
• The VkCommandPool that commandBuffer was allocated from, in vkCmdResetQueryPool
• The VkCommandPool that commandBuffer was allocated from, in vkCmdWriteTimestamp
• The VkCommandPool that commandBuffer was allocated from, in vkCmdCopyQueryPoolResults
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPushConstants
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass
• The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndRenderPass
• The VkCommandPool that commandBuffer was allocated from, in vkCmdExecuteCommands
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDeviceMask
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatchBase
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginVideoCodingKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdEndVideoCodingKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdControlVideoCodingKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDecodeVideoKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdSetDeviceMaskKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdDispatchBaseKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPushDescriptorSetKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdPushDescriptorSetWithTemplateKHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdBeginRenderPass2KHR
• The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass2
• The VkCommandPool that commandBuffer was allocated from, in vkCmdNextSubpass2KHR
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndRenderPass2`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndRenderPass2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirectCount`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndirectCountKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexedIndirectCount`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDrawIndexedIndirectCountKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetFragmentShadingRateKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEncodeVideoKHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdSetEvent2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdResetEvent2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWaitEvents2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdPipelineBarrier2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWriteTimestamp2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdWriteBufferMarker2AMD`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyBuffer2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyImage2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyBufferToImage2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdCopyImageToBuffer2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBlitImage2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdResolveImage2KHR`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDebugMarkerBeginEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDebugMarkerEndEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdDebugMarkerInsertEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBindTransformFeedbackBuffersEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdBeginTransformFeedbackEXT`
- The `VkCommandPool` that `commandBuffer` was allocated from, in `vkCmdEndTransformFeedbackEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdBeginQueryIndexedEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdEndQueryIndexedEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdDrawIndirectByteCountEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdEndQueryIndexedEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdDrawIndirectCountAMD`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdDrawIndexedIndirectCountAMD`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdBeginConditionalRenderingEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdEndConditionalRenderingEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetViewportWScalingNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetDiscardRectangleEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdBeginDebugUtilsLabelEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdEndDebugUtilsLabelEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdInsertDebugUtilsLabelEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetSampleLocationsEXT`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdBindShadingRateImageNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetViewportShadingRatePaletteNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetCoarseSampleOrderNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdBuildAccelerationStructureNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdCopyAccelerationStructureNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdTraceRaysNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdWriteAccelerationStructuresPropertiesNV`
• The VkCommandPool that commandBuffer was allocated from, in `vkCmdWriteBufferMarkerAMD`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdDrawMeshTasksNV`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdDrawMeshTasksIndirectNV`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdDrawMeshTasksIndirectCountNV`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetExclusiveScissorNV`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetCheckpointNV`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetPerformanceMarkerINTEL`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetPerformanceStreamMarkerINTEL`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetPerformanceOverrideINTEL`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetLineStippleEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetCullModeEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetFrontFaceEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetPrimitiveTopologyEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetViewportWithCountEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetScissorWithCountEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdBindVertexBuffers2EXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetDepthTestEnableEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetDepthWriteEnableEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetDepthCompareOpEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetDepthBoundsTestEnableEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetStencilTestEnableEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdSetStencilOpEXT`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdPreprocessGeneratedCommandsNV`
- The VkCommandPool that commandBuffer was allocated from, in `vkCmdExecuteGeneratedCommandsNV`
• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdBindPipelineShaderGroupNV

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetFragmentShadingRateEnumNV

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetVertexInputEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetPatchControlPointsEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetRasterizerDiscardEnableEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetDepthBiasEnableEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetLogicOpEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetPrimitiveRestartEnableEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetColorWriteEnableEXT

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdBuildAccelerationStructuresKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdBuildAccelerationStructuresIndirectKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdCopyAccelerationStructureKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdCopyAccelerationStructureToMemoryKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdCopyMemoryToAccelerationStructureKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdWriteAccelerationStructuresPropertiesKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdTraceRaysKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdTraceRaysIndirectKHR

• The VkCommandPool that commandBuffer was allocated from, in 
vkCmdSetRayTracingPipelineStackSizeKHR

3.7. Errors

Vulkan is a layered API. The lowest layer is the core Vulkan layer, as defined by this Specification. The application can use additional layers above the core for debugging, validation, and other purposes.

One of the core principles of Vulkan is that building and submitting command buffers should be
highly efficient. Thus error checking and validation of state in the core layer is minimal, although more rigorous validation can be enabled through the use of layers.

The core layer assumes applications are using the API correctly. Except as documented elsewhere in the Specification, the behavior of the core layer to an application using the API incorrectly is undefined, and may include program termination. However, implementations must ensure that incorrect usage by an application does not affect the integrity of the operating system, the Vulkan implementation, or other Vulkan client applications in the system. In particular, any guarantees made by an operating system about whether memory from one process can be visible to another process or not must not be violated by a Vulkan implementation for any memory allocation. Vulkan implementations are not required to make additional security or integrity guarantees beyond those provided by the OS unless explicitly directed by the application's use of a particular feature or extension.

Note
For instance, if an operating system guarantees that data in all its memory allocations are set to zero when newly allocated, the Vulkan implementation must make the same guarantees for any allocations it controls (e.g. VkDeviceMemory).

Similarly, if an operating system guarantees that use-after-free of host allocations will not result in values written by another process becoming visible, the same guarantees must be made by the Vulkan implementation for device memory.

If the protected memory feature is supported, the implementation provides additional guarantees when invalid usage occurs to prevent values in protected memory from being accessed or inferred outside of protected operations, as described in Protected Memory Access Rules.

Validation of correct API usage is left to validation layers. Applications should be developed with validation layers enabled, to help catch and eliminate errors. Once validated, released applications should not enable validation layers by default.

3.7.1. Valid Usage

Valid usage defines a set of conditions which must be met in order to achieve well-defined runtime behavior in an application. These conditions depend only on Vulkan state, and the parameters or objects whose usage is constrained by the condition.

Some valid usage conditions have dependencies on runtime limits or feature availability. It is possible to validate these conditions against Vulkan's minimum supported values for these limits and features, or some subset of other known values.

Valid usage conditions do not cover conditions where well-defined behavior (including returning an error code) exists.

Valid usage conditions should apply to the command or structure where complete information about the condition would be known during execution of an application. This is such that a validation layer or linter can be written directly against these statements at the point they are specified.
Valid usage conditions are described in a block labelled “Valid Usage” following each command or structure they apply to.

### 3.7.2. Implicit Valid Usage

Some valid usage conditions apply to all commands and structures in the API, unless explicitly denoted otherwise for a specific command or structure. These conditions are considered *implicit*, and are described in a block labelled “Valid Usage (Implicit)” following each command or structure they apply to. Implicit valid usage conditions are described in detail below.

#### Valid Usage for Object Handles

Any input parameter to a command that is an object handle *must* be a valid object handle, unless otherwise specified. An object handle is valid if:

- It has been created or allocated by a previous, successful call to the API. Such calls are noted in the Specification.
- It has not been deleted or freed by a previous call to the API. Such calls are noted in the Specification.
- Any objects used by that object, either as part of creation or execution, *must* also be valid.

The reserved values `VK_NULL_HANDLE` and `NULL` can be used in place of valid non-dispatchable handles and dispatchable handles, respectively, when *explicitly called out in the Specification*. Any command that creates an object successfully *must* not return these values. It is valid to pass these values to `vkDestroy*` or `vkFree*` commands, which will silently ignore these values.

#### Valid Usage for Pointers

Any parameter that is a pointer *must* be a *valid pointer* only if it is explicitly called out by a Valid Usage statement.

A pointer is “valid” if it points at memory containing values of the number and type(s) expected by the command, and all fundamental types accessed through the pointer (e.g. as elements of an array or as members of a structure) satisfy the alignment requirements of the host processor.
Valid Usage for Strings

Any parameter that is a pointer to char must be a finite sequence of values terminated by a null character, or if explicitly called out in the Specification, can be NULL.

Valid Usage for Enumerated Types

Any parameter of an enumerated type must be a valid enumerant for that type. A enumerant is valid if:

- The enumerant is defined as part of the enumerated type.
- The enumerant is not the special value (suffixed with \_MAX\_ENUM) defined for the enumerated type.

1

This special value exists only to ensure that C enum types are 32 bits in size. It is not part of the API, and should not be used by applications.

Any enumerated type returned from a query command or otherwise output from Vulkan to the application must not have a reserved value. Reserved values are values not defined by any extension for that enumerated type.

Note

This language is intended to accommodate cases such as “hidden” extensions known only to driver internals, or layers enabling extensions without knowledge of the application, without allowing return of values not defined by any extension.

Note

Application developers are encouraged to be careful when using switch statements with Vulkan API enums. This is because new extensions can add new values to existing enums. The use of a default: statement, within a switch, may avoid future compilation issues. Particularly for enums like e.g. VkDriverId that may change at any time.

Valid Usage for Flags

A collection of flags is represented by a bitmask using the type VkFlags:

```
// Provided by VK_VERSION_1_0
typedef uint32_t VkFlags;
```

Bitmasks are passed to many commands and structures to compactly represent options, but VkFlags is not used directly in the API. Instead, a Vk*Flags type which is an alias of VkFlags, and whose name matches the corresponding Vk*FlagBits that are valid for that type, is used.

Any Vk*Flags member or parameter used in the API as an input must be a valid combination of bit flags. A valid combination is either zero or the bitwise OR of valid bit flags. A bit flag is valid if:
• The bit flag is defined as part of the `Vk*FlagBits` type, where the bits type is obtained by taking the flag type and replacing the trailing `Flags` with `FlagBits`. For example, a flag value of type `VkColorComponentFlags` must contain only bit flags defined by `VkColorComponentFlagBits`.

• The flag is allowed in the context in which it is being used. For example, in some cases, certain bit flags or combinations of bit flags are mutually exclusive.

Any `Vk*Flags` member or parameter returned from a query command or otherwise output from Vulkan to the application may contain bit flags undefined in its corresponding `Vk*FlagBits` type. An application cannot rely on the state of these unspecified bits.

Only the low-order 31 bits (bit positions zero through 30) are available for use as flag bits.

**Note**

This restriction is due to poorly defined behavior by C compilers given a C enumerant value of `0x80000000`. In some cases adding this enumerant value may increase the size of the underlying `Vk*FlagBits` type, breaking the ABI.

**Valid Usage for Structure Types**

Any parameter that is a structure containing a `sType` member must have a value of `sType` which is a valid `VkStructureType` value matching the type of the structure.

Structure types supported by the Vulkan API include:

```c
// Provided by VK_VERSION_1_0
typedef enum VkStructureType {
    VK_STRUCTURE_TYPE_APPLICATION_INFO = 0,
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO = 1,
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO = 2,
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO = 3,
    VK_STRUCTURE_TYPE_SUBMIT_INFO = 4,
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO = 5,
    VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE = 6,
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO = 7,
    VK_STRUCTURE_TYPE_FENCE_CREATE_INFO = 8,
    VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO = 9,
    VK_STRUCTURE_TYPE_EVENT_CREATE_INFO = 10,
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO = 11,
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO = 12,
    VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO = 13,
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO = 14,
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO = 15,
    VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO = 16,
    VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO = 17,
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO = 18,
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO = 19,
    VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO = 20,
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO = 21,
    VK_STRUCTURE_TYPE_PIPELINE/browser/chapter3fundamentals.html#structureTypes
```
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO = 23,
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO = 24,
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO = 25,
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO = 26,
VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO = 27,
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO = 28,
VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO = 29,
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO = 30,
VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO = 31,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO = 32,
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO = 33,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO = 34,
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET = 35,
VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET = 36,
VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO = 37,
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO = 38,
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO = 39,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO = 40,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO = 41,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO = 42,
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO = 43,
VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER = 44,
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER = 45,
VK_STRUCTURE_TYPE_MEMORY_BARRIER = 46,
VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO = 47,
VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO = 48,

// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES = 1000094000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO = 1000157000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO = 1000157001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES = 1000083000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS = 1000127000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO = 1000127001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO = 1000060000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO = 1000060003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO = 1000060004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO = 1000060005,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_ALLOCATE_INFO = 1000060006,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO = 1000060007,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO = 1000060013,
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO = 1000060014,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES = 1000070000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO = 1000070001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2 = 1000146000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2 = 1000146001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2 = 1000146002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2 = 1000146003,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2 = 1000146004,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2 = 1000059000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2 = 1000059001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2 = 1000059002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2 = 1000059003,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2 = 1000059004,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2 = 1000059005,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2 = 1000059006,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2 = 1000059007,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2 = 1000059008,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES = 1000117000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO = 1000117001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO = 1000117002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO = 1000117003,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO = 1000053000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES = 1000053001,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES = 1000053002,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES = 1000120000,
  // Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO = 1000145000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES = 1000145001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES = 1000145002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPEDEVICE_QUEUE_INFO_2 = 1000145003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO = 1000156000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO = 1000156001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO = 1000156002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO = 1000156003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES = 1000156004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES = 1000156005,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO = 1000085000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO = 1000071000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES = 1000071001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO = 1000071002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES = 1000071003,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES = 1000071004,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO = 1000072000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO = 1000072001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO = 1000072002,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO = 1000112000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES = 1000112001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO = 1000113000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO = 1000077000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO = 1000076000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES = 1000076001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES = 1000168000,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT = 1000168001,
// Provided by VK_VERSION_1_1
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES = 1000063000,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO = 1000147000,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2 = 1000109000,
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2 = 1000109001,
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2 = 1000109002,
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2 = 1000109003,
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2 = 1000109004,
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO = 1000109005,
VK_STRUCTURE_TYPE_SUBPASS_END_INFO = 1000109006,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES = 1000177000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES = 1000196000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES = 1000180000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES = 1000082000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES = 1000197000,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO = 1000161000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES = 1000161001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES = 1000161002,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO = 1000161003,
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT = 1000161004,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES = 1000199000,
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE = 1000199001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES = 1000221000,
VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO = 1000246000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES = 1000130000,
VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO = 1000130001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES = 1000211000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES = 1000261000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES = 1000207000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES = 1000207001,
VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO = 1000207002,
VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO = 1000207003,
VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO = 1000207004,
VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO = 1000207005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES = 1000257000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES = 1000253000,
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT = 1000241001,
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT = 1000241002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES = 1000261000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES = 1000207000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES = 1000207001,
VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO = 1000207002,
VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO = 1000207003,
VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO = 1000207004,
VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO = 1000207005,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES = 1000257000,
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO = 1000244001,
VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO = 1000257002,
VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO = 1000257003,
VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO = 1000257004,
// Provided by VK_KHR_swapchain
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR = 1000001000,
// Provided by VK_KHR_swapchain
VK_STRUCTURE_TYPE_PRESENT_INFO_KHR = 1000001001,
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_surface
VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR = 1000060007,
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR = 1000060008,
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR = 1000060009,
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR = 1000060010,
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR = 1000060011,
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR = 1000060012,
// Provided by VK_KHR_display
VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR = 1000002000,
// Provided by VK_KHR_display
VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR = 1000002001,
// Provided by VK_KHR_xlib_surface
VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR = 1000004000,
// Provided by VK_KHR_xcb_surface
VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR = 1000005000,
// Provided by VK_KHR_wayland_surface
VK_STRUCTURE_TYPE_WAYLAND_SURFACE_CREATE_INFO_KHR = 1000006000,
// Provided by VK_KHR_android_surface
VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR = 1000008000,
// Provided by VK_KHR_win32_surface
VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR = 1000009000,
// Provided by VK_EXT_debug_report
VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT = 1000011000,
// Provided by VK_AMD_rasterization_order
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_RASTERIZATION_ORDER_AMD = 1000018000,
// Provided by VK_EXT_debug_marker
VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT = 1000022000,
// Provided by VK_EXT_debug_marker
VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_TAG_INFO_EXT = 1000022001,
VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT = 1000022002,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_PROFILE_KHR = 1000023000,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_CAPABILITIES_KHR = 1000023001,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_PICTURE_RESOURCE_KHR = 1000023002,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_GET_MEMORY_PROPERTIES_KHR = 1000023003,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_BIND_MEMORY_KHR = 1000023004,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_SESSION_CREATE_INFO_KHR = 1000023005,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_SESSION_PARAMETERS_CREATE_INFO_KHR = 1000023006,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_SESSION_PARAMETERS_UPDATE_INFO_KHR = 1000023007,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_BEGIN CODING_INFO_KHR = 1000023008,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_END CODING_INFO_KHR = 1000023009,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_CODING_CONTROL_INFO_KHR = 1000023010,
#endif VK_ENABLE_BETA_EXTENSIONS

// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_REFERENCE_SLOT_KHR = 1000023011,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_QUEUE_FAMILY_PROPERTIES_2_KHR = 1000023012,
#endif
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_VIDEO_PROFILES_KHR = 1000023013,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_queue
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VIDEO_FORMAT_INFO_KHR = 1000023014,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_encode_queue
VK_STRUCTURE_TYPE_VIDEO_ENCODE_INFO_KHR = 1000299000,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_encode_queue
VK_STRUCTURE_TYPE_VIDEO_ENCODE_RATE_CONTROL_INFO_KHR = 1000299001,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_NV_dedicated_allocation
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV = 1000026000,
// Provided by VK_NV_dedicated_allocation
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_BUFFER_CREATE_INFO_NV = 1000026001,
// Provided by VK_NV_dedicated_allocation
VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV = 1000026002,
// Provided by VK_EXT_transform_feedback
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_FEATURES_EXT = 1000028000,
// Provided by VK_EXT_transform_feedback
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_PROPERTIES_EXT = 1000028001,
// Provided by VK_EXT_transform_feedback
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_STREAM_CREATE_INFO_EXT = 1000028002,
// Provided by VK_NVX_image_view_handle
VK_STRUCTURE_TYPE_IMAGE_VIEW_HANDLE_INFO_NVX = 1000030000,
// Provided by VK_NVX_image_view_handle
VK_STRUCTURE_TYPE_IMAGE_VIEW_ADDRESS_PROPERTIES_NVX = 1000030001,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_EXT_video_encode_h264
VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_CAPABILITIES_EXT = 1000038000,
#endif VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_EXT_video_encode_h264
VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_CREATE_INFO_EXT = 1000038001,
#endif
#elif defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_PARAMETERS_CREATE_INFO_EXT = 1000038002,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_PARAMETERS_ADD_INFO_EXT = 1000038003,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_VCL_FRAME_INFO_EXT = 1000038004,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_DPB_SLOT_INFO_EXT = 1000038005,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_NALU_SLICE_EXT = 1000038006,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264.Emit_PICTURE_PARAMETERS_EXT = 1000038007,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_PROFILE_EXT = 1000038008,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h264
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_CAPABILITIES_EXT = 1000040000,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h264
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_CREATE_INFO_EXT = 1000040001,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h264
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_PICTURE_INFO_EXT = 1000040002,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h264
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_MVC_EXT = 1000040003,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h264
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_PROFILE_EXT = 1000040004,
// Provided by VK_EXT_video_decode_h264
VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_PARAMETERS_CREATE_INFO_EXT = 1000040005,
#endif
#elif USE_H264

// Provided by VK_EXT_video_decode_h264
VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_PARAMETERS_ADD_INFO_EXT = 1000040006,
#endif

// Provided by VK_EXT_video_decode_h264
VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_DPB_SLOT_INFO_EXT = 1000040007,
#endif

// Provided by VK_AMD_texture_gather_bias_lod
VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD = 1000041000,
// Provided by VK_GGP_stream_descriptor_surface
VK_STRUCTURE_TYPE_STREAM_DESCRIPTOR_SURFACE_CREATE_INFO_GGP = 1000049000,
// Provided by VK_AMD_texture_gather_bias_lod
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CORNER_SAMPLED_IMAGE_FEATURES_NV = 1000050000,
// Provided by VK_NV_external_memory
VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV = 1000056000,
// Provided by VK_NV_external_memory
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV = 1000056001,
// Provided by VK_NV_external_memory_win32
VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_NV = 1000057000,
// Provided by VK_NV_external_memory_win32
VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV = 1000057001,
// Provided by VK_NV_win32_keyed_mutex
VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV = 1000058000,
// Provided by VK_EXT_validation_flags
VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT = 1000061000,
// Provided by VK_NN_vi_surface
VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN = 1000062000,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT = 1000066000,
// Provided by VK_EXT_astc_decode_mode
VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT = 1000067000,
// Provided by VK_EXT_astc_decode_mode
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT = 1000067001,
// Provided by VK_KHR_external_memory_win32
VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHR = 1000073000,
// Provided by VK_KHR_external_memory_win32
VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR = 1000073001,
// Provided by VK_KHR_external_memory_win32
VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHR = 1000073002,
// Provided by VK_KHR_external_memory_win32
VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR = 1000073003,
// Provided by VK_KHR_external_memory_fd
VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHR = 1000074000,
// Provided by VK_KHR_external_memory_fd
VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR = 1000074001,
// Provided by VK_KHR_external_memory_fd
VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR = 1000074002,
// Provided by VK_KHR_win32_keyed_mutex
VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHR = 1000075000,
// Provided by VK_KHR_external_semaphore_win32
VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR = 1000078000,
// Provided by VK_KHR_external_semaphore_win32
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR = 1000078001,
// Provided by VK_KHR_external_semaphore_win32
VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHR = 1000078002,
// Provided by VK_KHR_external_semaphore_win32
VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR = 1000078003,
// Provided by VK_KHR_external_semaphore_fd
VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR = 1000079000,
// Provided by VK_KHR_external_semaphore_fd
VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR = 1000079001,
// Provided by VK_KHR_push_descriptor
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PUSH_DESCRIPTOR_PROPERTIES_KHR = 1000080000,
// Provided by VK_EXT_conditional_rendering
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_CONDITIONAL_RENDERING_INFO_EXT = 1000081000,
// Provided by VK_KHR_incremental_present
VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR = 1000084000,
// Provided by VK_NV_clip_space_w_scaling
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV = 1000087000,
// Provided by VK_EXT_display_surface_counter
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT = 1000090000,
// Provided by VK_EXT_display_control
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PER_VIEW_ATTRIBUTES_PROPERTIES_NVX = 1000097000,
// Provided by VK_NV_viewport_swizzle
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SWIZZLE_STATE_CREATE_INFO_NV = 1000098000,
// Provided by VK_EXT_discard_rectangles
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT = 1000099000,
// Provided by VK_EXT_discard_rectangles
VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT = 1000099001,
// Provided by VK_EXT_conservative_rasterization
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_KHR = 1000100000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT = 1000101000,
   // Provided by VK_EXT_conservative_rasterization
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT = 1000101001,
   // Provided by VK_EXT_depth_clip_enable
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLE_FEATURES_EXT = 1000102000,
   // Provided by VK_EXT_depth_clip_enable
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT = 1000102001,
   // Provided by VK_EXT_hdr_metadata
VK_STRUCTURE_TYPE_HDR_METADATA_EXT = 1000105000,
   // Provided by VK_KHR_shared_presentable_image
VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR = 1000111000,
   // Provided by VK_KHR_external_fence_win32
VK_STRUCTURE_TYPE_IMPORT_FENCE_WIN32_HANDLE_INFO_KHR = 1000114000,
   // Provided by VK_KHR_external_fence_win32
VK_STRUCTURE_TYPE_EXPORT_FENCE_WIN32_HANDLE_INFO_KHR = 1000114001,
   // Provided by VK_KHR_external_fence_win32
VK_STRUCTURE_TYPE_FENCE_GET_WIN32_HANDLE_INFO_KHR = 1000114002,
   // Provided by VK_KHR_external_fence_fd
VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR = 1000115000,
   // Provided by VK_KHR_external_fence_fd
VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR = 1000115001,
   // Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR = 1000116000,
   // Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR = 1000116001,
   // Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR = 1000116002,
   // Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR = 1000116003,
   // Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR = 1000116005,
   // Provided by VK_KHR_performance_query
VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR = 1000116006,
   // Provided by VK_KHR_get_surface_capabilities2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR = 1000119000,
   // Provided by VK_KHR_get_surface_capabilities2
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR = 1000119001,
   // Provided by VK_KHR_get_surface_capabilities2
VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR = 1000119002,
   // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR = 1000121000,
   // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR = 1000121001,
   // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR = 1000121002,
   // Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR = 1000121003,
// Provided by VK_KHR_get_display_properties2
VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR = 1000121004,
// Provided by VK_MVK_ios_surface
VK_STRUCTURE_TYPE_IOS_SURFACE_CREATE_INFO_MVK = 1000122000,
// Provided by VK_MVK_macos_surface
VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK = 1000123000,
// Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT = 1000128000,
// Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT = 1000128001,
// Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT = 1000128002,
// Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT = 1000128003,
// Provided by VK_EXT_debug_utils
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT = 1000128004,
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_USAGE_ANDROID = 1000129000,
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_PROPERTIES_ANDROID = 1000129001,
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_FORMAT_PROPERTIES_ANDROID = 1000129002,
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VK_STRUCTURE_TYPE_IMPORT_ANDROID_HARDWARE_BUFFER_INFO_ANDROID = 1000129003,
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VK_STRUCTURE_TYPE_MEMORY_GET_ANDROID_HARDWARE_BUFFER_INFO_ANDROID = 1000129004,
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VK_STRUCTURE_TYPE_EXTERNAL_FORMAT_ANDROID = 1000129005,
// Provided by VK_EXT_inline_uniform_block
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES_EXT = 1000138000,
// Provided by VK_EXT_inline_uniform_block
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES_EXT = 1000138001,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT = 1000138002,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT = 1000138003,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT = 1000143000,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT = 1000143001,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT = 1000143002,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_PHYSICAL DEVICE_INLINE_UNIFORM BLOCK FEATURES_EXT = 1000138000,
// Provided by VK_EXT_inline_uniform_block
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT = 1000138002,
// Provided by VK_EXT_inline_uniform_block
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT = 1000138003,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT = 1000143000,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT = 1000143001,
// Provided by VK_EXT_sample_locations
VK_STRUCTURE_TYPE_PHYSICAL DEVICE_INLINE_UNIFORM BLOCK FEATURES_EXT = 1000138000,
// Provided by VK_EXT_inline_uniform_block
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT = 1000138002,
// Provided by VK_EXT_inline_uniform_block
VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM BLOCK_CREATE_INFO_EXT = 1000138003,
// Provided by VK_EXT_blend_operation_advanced
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT = 1000148000,

// Provided by VK_EXT_blend_operation_advanced
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT = 1000148001,

// Provided by VK_NV_fragment_coverage_to_color
VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_TO_COLOR_STATE_CREATE_INFO_NV = 1000149000,

// Provided by VK_KHR_acceleration_structure
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR = 1000150007,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR = 1000150000,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR = 1000150002,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_AABBS_DATA_KHR = 1000150003,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR = 1000150004,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR = 1000150005,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR = 1000150006,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_VERSION_INFO_KHR = 1000150009,
VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_INFO_KHR = 1000150010,
VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_TO_MEMORY_INFO_KHR = 1000150011,
VK_STRUCTURE_TYPE_COPY_MEMORY_TO_ACCELERATION_STRUCTURE_INFO_KHR = 1000150012,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR = 1000150013,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_PROPERTIES_KHR = 1000150014,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR = 1000150017,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR = 1000150020,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR = 1000347000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR = 1000347001,
VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR = 1000150015,
VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR = 1000150016,
VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_INTERFACE_CREATE_INFO_KHR = 1000150018,
// Provided by VK_KHR_ray_query
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_QUERY_FEATURES_KHR = 1000348013,
// Provided by VK_NV_framebuffer_mixed_samples
VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_MODULATION_STATE_CREATE_INFO_NV = 1000152000,
// Provided by VK_NV_shader_sm_builtins
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_FEATURES_NV = 1000154000,
// Provided by VK_NV_shader_sm_builtins
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_PROPERTIES_NV = 1000154001,
// Provided by VK_EXT_image_drm_format_modifer
VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT = 1000158000,
// Provided by VK_EXT_image_drm_format_modifer
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT = 1000158002,
// Provided by VK_EXT_image_drm_format_modifer
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT = 1000158003,
// Provided by VK_EXT_image_drm_format_modifer
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT = 1000158004,
// Provided by VK_EXT_image_drm_format_modifer
VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT = 1000158005,
// Provided by VK_EXT_validation_cache
VK_STRUCTURE_TYPE_VALIDATION_CACHE_CREATE_INFO_EXT = 1000160000,
// Provided by VK_EXT_validation_cache
VK_STRUCTURE_TYPE_SHADER_MODULE_VALIDATION_CACHE_CREATE_INFO_EXT = 1000160001,
// Provided by VK_KHR_portability_subset
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PORTABILITY_SUBSET_FEATURES_KHR = 1000163000,
// Provided by VK_KHR_portability_subset
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PORTABILITY_SUBSET_PROPERTIES_KHR = 1000163001,
// Provided by VK_KHR_portability_subset
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SHADING_RATE_IMAGE_STATE_CREATE_INFO_NV = 1000164000,
// Provided by VK_NV_shading_rate_image
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_FEATURES_NV = 1000164001,
// Provided by VK_NV_shading_rate_image
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_PROPERTIES_NV = 1000164002,
// Provided by VK_NV_shading_rate_image
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_COARSE_SAMPLE_ORDER_STATE_CREATE_INFO_NV = 1000164005,
// Provided by VK_NV_ray_tracing
VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_NV = 1000165000,
// Provided by VK_NV_ray_tracing
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV = 1000165001,
// Provided by VK_NV_ray_tracing
VK_STRUCTURE_TYPE_GEOMETRY_NV = 1000165003,
// Provided by VK_NV_ray_tracing
VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_NV = 1000165004,
// Provided by VK_NV_ray_tracing
VK_STRUCTURE_TYPE_GEOMETRY_AABB_NV = 1000165005,
VK_STRUCTURE_TYPE_BIND_ACCELERATION_STRUCTURE_MEMORY_INFO_NV = 1000165006,
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_NV = 1000165007,
VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_INFO_NV = 1000165008,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICERAY TRACING_PROPERTIES_NV = 1000165009,
VK_STRUCTURE_TYPERAY TRACING_SHADER_GROUP_CREATE_INFO_NV = 1000165011,
VK_STRUCTURE_TYPEACCELERATION_STRUCTURE_INFO_NV = 1000165012,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_REPRESENTATIVE_FRAGMENT_TEST_FEATURES_NV = 1000166000,
VK_STRUCTURE_TYPE_PIPELINE_REPRESENTATIVE_FRAGMENT_TEST_STATE_CREATE_INFO_NV = 1000166001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT = 1000170000,
VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT = 1000170001,
VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT = 1000174000,
VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT = 1000178000,
VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT = 1000178001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT = 1000178002,
VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_CAPABILITIES_EXT = 1000187000,
VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_CREATE_INFO_EXT = 1000187001,
VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_PARAMETERS_CREATE_INFO_EXT = 1000187002,
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h265
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_PARAMETERS_ADD_INFO_EXT = 1000187003,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h265
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_PROFILE_EXT = 1000187004,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h265
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_PICTURE_INFO_EXT = 1000187005,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h265
    VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_DPB_SLOT_INFO_EXT = 1000187006,
#endif

// Provided by VK_AMD_memory_overallocation_behavior
VK_STRUCTURE_TYPE_DEVICE_MEMORY_OVERALLOCATION_CREATE_INFO_AMD = 1000189000,

// Provided by VK_EXT_vertex_attribute_divisor
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_PROPERTIES_EXT = 1000190000,

// Provided by VK_EXT_vertex_attribute_divisor
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT = 1000190001,

// Provided by VK_EXT_vertex_attribute_divisor
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT = 1000190002,

// Provided by VK_GGP_frame_token
VK_STRUCTURE_TYPE_PRESENT_FRAME_TOKEN_GGP = 1000191000,

// Provided by VK_EXT_pipeline_creation_feedback
VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO_EXT = 1000192000,

// Provided by VK_NV_compute_shader_derivatives
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COMPUTE_SHADER_DERIVATIVES_FEATURES_NV = 1000201000,

// Provided by VK_NV_mesh_shader
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_FEATURES_NV = 1000202000,

// Provided by VK_NV_mesh_shader
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_PROPERTIES_NV = 1000202001,

// Provided by VK_NV_fragment_shader_barycentric
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_BARYCENTRIC_FEATURES_NV = 1000203000,

// Provided by VK_NV_shader_image_footprint
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_FOOTPRINT_FEATURES_NV = 1000204000,

// Provided by VK_NV_scissor_exclusive
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_EXCLUSIVE_SCISSOR_STATE_CREATE_INFO_NV = 1000205000,

// Provided by VK_NV_scissor_exclusive
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXCLUSIVE_SCISSOR_FEATURES_NV = 1000205002,

// Provided by VK_NV_device_diagnostic_checkpoints
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DIAGNOSTIC_CHECKPOINTS_FEATURES_NV = 1000206000,
// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_PROPERTIES_KHR = 1000226002,

// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_FEATURES_KHR = 1000226003,

// Provided by VK_KHR_fragment_shading_rate
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_KHR = 1000226004,

// Provided by VK_AMD_shader_core_properties2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_2_AMD = 1000227000,

// Provided by VK_AMD_device_coherent_memory
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COHERENT_MEMORY_FEATURES_AMD = 1000229000,

// Provided by VK_EXT_shader_image_atomic_int64
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_ATOMIC_INT64_FEATURES_EXT = 1000234000,

// Provided by VK_EXT_memory_budget
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT = 1000237000,

// Provided by VK_EXT_memory_priority
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PRIORITY_FEATURES_EXT = 1000238000,

// Provided by VK_KHR_surface_protected_capabilities
VK_STRUCTURE_TYPE_SURFACE_PROTECTED_CAPABILITIES_KHR = 1000239000,

// Provided by VK_NV_dedicated_allocation_image_aliasing
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEDICATED_ALLOCATION_IMAGE_ALIASING_FEATURES_NV = 1000240000,

// Provided by VK_EXT_buffer_device_address
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT = 1000244000,

// Provided by VK_EXT_buffer_device_address
VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_CREATE_INFO_EXT = 1000244002,

// Provided by VK_EXT_tooling_info
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TOOL_PROPERTIES_EXT = 1000245000,

// Provided by VK_EXT_validation_features
VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT = 1000247000,

// Provided by VK_NV_cooperative_matrix
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_PROPERTIES_NV = 1000249001,

// Provided by VK_NV_cooperative_matrix
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_FEATURES_NV = 1000249002,

// Provided by VK_NV_coverage_reduction_mode
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COVERAGE_REDUCTION_MODE_FEATURES_NV = 1000250000,

// Provided by VK_NV_coverage_reduction_mode
VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_REDUCTION_STATE_CREATE_INFO_NV = 1000250001,

// Provided by VK_NV_coverage_reduction_mode
VK_STRUCTURE_TYPE_FRAMEBUFFER_MIXED_SAMPLES_COMBINATION_NV = 1000250002,

// Provided by VK_EXT_fragment_shader_interlock
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT = 1000251000,

// Provided by VK_EXT_ycbcr_image_arrays
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT = 1000252000,
// Provided by VK_EXT_provoking_vertex
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROVOKING_VERTEX_FEATURES_EXT = 1000254000,
// Provided by VK_EXT_provoking_vertex
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_PROVOKING_VERTEX_STATE_CREATE_INFO_EXT = 1000254001,
// Provided by VK_EXT_full_screen_exclusive
VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_INFO_EXT = 1000255000,
// Provided by VK_EXT_full_screen_exclusive
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_FULL_SCREEN_EXCLUSIVE_EXT = 1000255002,
// Provided by VK_EXT_full_screen_exclusive with VK_KHR_win32_surface
VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_WIN32_INFO_EXT = 1000255001,
// Provided by VK_EXT_headless_surface
VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT = 1000256000,
// Provided by VK_EXT_line_rasterization
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT = 1000259000,
// Provided by VK_EXT_line_rasterization
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT = 1000259001,
// Provided by VK_EXT_line_rasterization
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT = 1000259002,
// Provided by VK_EXT_shader_atomic_float
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT = 1000260000,
// Provided by VK_EXT_index_type_uint8
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT = 1000265000,
// Provided by VK_EXT_extended_dynamic_state
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_FEATURES_EXT = 1000267000,
// Provided by VK_KHR_pipeline_executable_properties
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_EXECUTABLE_PROPERTIES_FEATURES_KHR = 1000269000,
// Provided by VK_KHR_pipeline_executable_properties
VK_STRUCTURE_TYPE_PIPELINE_INFO_KHR = 1000269001,
// Provided by VK_KHR_pipeline_executable_properties
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_PROPERTIES_KHR = 1000269002,
// Provided by VK_KHR_pipeline_executable_properties
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INFO_KHR = 1000269003,
// Provided by VK_KHR_pipeline_executable_properties
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_STATISTIC_KHR = 1000269004,
// Provided by VK_KHR_pipeline_executable_properties
VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INTERNAL_REPRESENTATION_KHR = 1000269005,
// Provided by VK_EXT_shader_demote_to_helper_invocation
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT = 1000276000,
// Provided by VK_NV_device_generated_commands
VK_STRUCTURE_TYPE_GRAPHICS_SHADER_GROUP_CREATE_INFO_NV = 1000277001,
// Provided by VK_NV_device_generated_commands
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_SHADER_GROUPS_CREATE_INFO_NV = 1000277002,
// Provided by VK_NV_device_generated_commands
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_PROPERTIES_NV = 1000277000,
VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_TOKEN_NV = 1000277003,
VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_CREATE_INFO_NV = 1000277004,
VK_STRUCTURE_TYPE_GENERATED_COMMANDS_INFO_NV = 1000277005,
VK_STRUCTURE_TYPE_GENERATED_COMMANDS_MEMORY_REQUIREMENTS_INFO_NV = 1000277006,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_FEATURES_NV = 1000277007,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INHERITED_VIEWPORT_SCISSOR_FEATURES_NV = 1000278000,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_VIEWPORT_SCISSOR_INFO_NV = 1000278001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT = 1000281000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT = 1000281001,
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_RENDER_PASS_TRANSFORM_INFO_QCOM = 1000282000,
VK_STRUCTURE_TYPE_RENDER_PASS_TRANSFORM_BEGIN_INFO_QCOM = 1000282001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_MEMORY_REPORT_FEATURES_EXT = 1000284000,
VK_STRUCTURE_TYPE_DEVICE_DEVICE_MEMORY_REPORT_CREATE_INFO_EXT = 1000284001,
VK_STRUCTURE_TYPE_DEVICE_MEMORY_REPORT_CALLBACK_DATA_EXT = 1000284002,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_FEATURES_EXT = 1000286000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_PROPERTIES_EXT = 1000286001,
VK_STRUCTURE_TYPE_SAMPLER_CUSTOM_BORDER_COLOR_CREATE_INFO_EXT = 1000287000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_PROPERTIES_EXT = 1000287001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_FEATURES_EXT = 1000287002,
VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR = 1000290000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PRIVATE_DATA_FEATURES_EXT = 1000295000,
VK_STRUCTURE_TYPE_DEVICE_PRIVATE_DATA_CREATE_INFO_EXT = 1000295001,
VK_STRUCTURE_TYPE_PRIVATE_DATA_SLOT_CREATE_INFO_EXT = 1000295002,

// Provided by VK_EXT_pipeline_creation_cache_control
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_CREATION_CACHE_CONTROL_FEATURES_EXT = 1000297000,

// Provided by VK_NV_device_diagnostics_config
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DIAGNOSTICS_CONFIG_FEATURES_NV = 1000300000,

// Provided by VK_NV_device_diagnostics_config
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DIAGNOSTICS_CONFIG_CREATE_INFO_NV = 1000300001,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_MEMORY_BARRIER_2_KHR = 1000314000,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2_KHR = 1000314001,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2_KHR = 1000314002,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_DEPENDENCY_INFO_KHR = 1000314003,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_SUBMIT_INFO_2_KHR = 1000314004,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO_KHR = 1000314005,

// Provided by VK_KHR_synchronization2
VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO_KHR = 1000314006,

// Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES_KHR = 1000314007,

// Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_2_NV = 1000314008,

// Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
VK_STRUCTURE_TYPE_CHECKPOINT_DATA_2_NV = 1000314009,

// Provided by VK_KHR_zero_initialize_workgroup_memory
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ZERO_INITIALIZE_WORKGROUP_MEMORY_FEATURES_KHR = 1000325000,

// Provided by VK_NV_fragment_shading_rate Enums
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_ENUMS_PROPERTIES_NV = 1000326000,

// Provided by VK_NV_fragment_shading_rate Enums
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_ENUMS_FEATURES_NV = 1000326001,

// Provided by VK_NV_fragment_shading_rate Enums
VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_ENUM_STATE_CREATE_INFO_NV = 1000326002,

// Provided by VK_EXT_ycbcr_2plane_444_formats
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_2_PLANE_444_FORMATS_FEATURES_EXT = 1000330000,

// Provided by VK_EXT_fragment_density_map2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_2_FEATURES_EXT = 1000332000,

// Provided by VK_EXT_fragment_density_map2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_2_PROPERTIES_EXT = 1000332001,

// Provided by VK_QCOM_rotated_copy_commands
VK_STRUCTURE_TYPE_COPY_COMMAND_TRANSFORM_INFO_QCOM = 1000333000,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES_EXT = 1000335000,

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_WORKGROUP_MEMORY_EXPLICIT_LAYOUT_FEATURES_KHR = 1000336000,

VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2_KHR = 1000337000,

VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2_KHR = 1000337001,

VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2_KHR = 1000337002,

VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2_KHR = 1000337003,

VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2_KHR = 1000337004,

VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2_KHR = 1000337005,

VK_STRUCTURE_TYPE_BUFFER_COPY_2_KHR = 1000337006,

VK_STRUCTURE_TYPE_IMAGE_COPY_2_KHR = 1000337007,

VK_STRUCTURE_TYPE_IMAGE_BLIT_2_KHR = 1000337008,

VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2_KHR = 1000337009,

VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2_KHR = 1000337010,

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_4444_FORMATS_FEATURES_EXT = 1000340000,

VK_STRUCTURE_TYPE_DIRECTFB_SURFACE_CREATE_INFO_EXT = 1000346000,

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MUTABLE_DESCRIPTOR_TYPE_FEATURES_VALVE = 1000351000,

VK_STRUCTURE_TYPE_MUTABLE_DESCRIPTOR_TYPE_CREATE_INFO_VALVE = 1000351002,

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_INPUT_DYNAMIC_STATE_FEATURES_EXT = 1000352000,

VK_STRUCTURE_TYPE_VERTEX_INPUT_BINDING_DESCRIPTION_2_EXT = 1000352001,

VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION_2_EXT = 1000352002,

VK_STRUCTURE_TYPE_IMPORT_MEMORY_ZIRCON_HANDLE_INFO_FUCHSIA = 1000364000,

VK_STRUCTURE_TYPE_MEMORY_ZIRCON_HANDLE_PROPERTIES_FUCHSIA = 1000364001,

VK_STRUCTURE_TYPE_MEMORY_GET_ZIRCON_HANDLE_INFO_FUCHSIA = 1000364002,

VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_ZIRCON_HANDLE_INFO_FUCHSIA = 1000365000.
// Provided by VK_FUCHSIA_external_semaphore
VK_STRUCTURE_TYPE_SEMAPHORE_GET_ZIRCON_HANDLE_INFO_FUCHSIA = 1000365001,
// Provided by VK_EXT_extended_dynamic_state2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_2_FEATURES_EXT = 1000377000,
// Provided by VK_QNX_screen_surface
VK_STRUCTURE_TYPE_SCREEN_SURFACE_CREATE_INFO_QNX = 1000378000,
// Provided by VK_EXT_color_write_enable
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COLOR_WRITE_ENABLE_FEATURES_EXT = 1000381000,
VK_STRUCTURE_TYPE_PIPELINE_COLOR_WRITE_CREATE_INFO_EXT = 1000381001,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETER_FEATURES = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES,
VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT = VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT,
// Provided by VK_KHR_multiview
VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO_KHR = VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO,
// Provided by VK_KHR_multiview
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES_KHR = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES,
// Provided by VK_KHR_multiview
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES_KHR = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES,
// Provided by VK_KHR_get_physical_device_properties2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2,
VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2,
VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2_KHR = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2,
VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2,
VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2_KHR = VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2,
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2,
// Provided by VK_KHR_device_group
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHR =
VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO,
// Provided by VK_KHR_device_group
VK_STRUCTURE_TYPEDEVICE_GROUP_RENDER_PASS_BEGIN_INFO_KHR =
VK_STRUCTURE_TYPEDEVICE_GROUP_RENDER_PASS_BEGIN_INFO,
// Provided by VK_KHR_device_group
VK_STRUCTURE_TYPEDEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO_KHR =
VK_STRUCTURE_TYPEDEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO,
// Provided by VK_KHR_device_group
VK_STRUCTURE_TYPEDEVICE_GROUP_SUBMIT_INFO_KHR =
VK_STRUCTURE_TYPEDEVICE_GROUP_SUBMIT_INFO,
// Provided by VK_KHR_device_group
VK_STRUCTURE_TYPEDEVICE_GROUP_BIND_SPARSE_INFO_KHR =
VK_STRUCTURE_TYPEDEVICE_GROUP_BIND_SPARSE_INFO,
// Provided by VK_KHR_device_group with VK_KHR_bind_memory2
VK_STRUCTURE_TYPEBIND_BUFFER_MEMORYDEVICE_GROUP_INFO_KHR =
VK_STRUCTURE_TYPEBIND_BUFFER_MEMORYDEVICE_GROUP_INFO,
// Provided by VK_KHR_device_group with VK_KHR_bind_memory2
VK_STRUCTURE_TYPEBIND_IMAGE_MEMORYDEVICE_GROUP_INFO_KHR =
VK_STRUCTURE_TYPEBIND_IMAGE_MEMORYDEVICE_GROUP_INFO,
// Provided by VK_KHR_device_group_creation
VK_STRUCTURE_TYPE_PHYSICALDEVICE_GROUP_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICALDEVICE_GROUP_PROPERTIES,
// Provided by VK_KHR_device_group_creation
VK_STRUCTURE_TYPEDEVICE_GROUP_DEVICE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPEDEVICE_GROUP_DEVICE_CREATE_INFO,
// Provided by VK_KHR_external_memory_capabilities
VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_IMAGE_FORMAT_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_IMAGE_FORMAT_INFO,
// Provided by VK_KHR_external_memory_capabilities
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES,
// Provided by VK_KHR_external_memory_capabilities
VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_BUFFER_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICALDEVICE_EXTERNAL_BUFFER_INFO,
// Provided by VK_KHR_external_memory_capabilities
VK_STRUCTURE_TYPE_EXTERNALBUFFER_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNALBUFFER_PROPERTIES,
// Provided by VK_KHR_external_memory_capabilities,
VK_KHR_external_semaphore_capabilities, VK_KHR_external_fence_capabilities
VK_STRUCTURE_TYPE_PHYSICALDEVICE_ID_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICALDEVICE_ID_PROPERTIES,
// Provided by VK_KHR_external_memory
VK_STRUCTURE_TYPE_EXTERNALMEMORY_BUFFER_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXTERNALMEMORY_BUFFER_CREATE_INFO,
// Provided by VK_KHR_external_memory
VK_STRUCTURE_TYPE_EXTERNALMEMORY_IMAGE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXTERNALMEMORY_IMAGE_CREATE_INFO,
// Provided by VK_KHR_external_memory
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO,
// Provided by VK_KHR_external_semaphore_capabilities
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO,
// Provided by VK_KHR_external_semaphore_capabilities
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES,
// Provided by VK_KHR_external_semaphore
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO,
// Provided by VK_KHR_shader_float16_int8
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES,
// Provided by VK_KHR_shader_float16_int8
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT16_INT8_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES,
// Provided by VK_KHR_16bit_storage
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES,
// Provided by VK_KHR_descriptor_update_template
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO,
// Provided by VK_KHR_imageless_framebuffer
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES2_EXT =
VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT,
// Provided by VK_KHR_imageless_framebuffer
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES,
// Provided by VK_KHR_imageless_framebuffer
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO,
// Provided by VK_KHR_imageless_framebuffer
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO_KHR =
VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2_KHR =
VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2_KHR =
VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2_KHR =
VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2_KHR =
VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2_KHR =
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO_KHR = VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO,
// Provided by VK_KHR_create_renderpass2
VK_STRUCTURE_TYPE_SUBPASS_END_INFO_KHR = VK_STRUCTURE_TYPE_SUBPASS_END_INFO,
// Provided by VK_KHR_external_fence_capabilities
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO,
// Provided by VK_KHR_external_fence_capabilities
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES,
// Provided by VK_KHR_external_fence
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO,
// Provided by VK_KHR_maintenance2
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES,
// Provided by VK_KHR_maintenance2
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO,
// Provided by VK_KHR_maintenance2
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO,
// Provided by VK_KHR_maintenance2
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO,
// Provided by VK_KHR_variable_pointers
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES,
// Provided by VK_KHR_variable_pointers
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTER_FEATURES,
// Provided by VK_KHR_dedicated_allocation
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS_KHR =
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS,
// Provided by VK_KHR_dedicated_allocation
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR =
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO,
// Provided by VK_KHR_dedicated_allocation
VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR,
// Provided by VK_KHR_dedicated_allocation
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2_KHR =
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2,
// Provided by VK_KHR_dedicated_allocation
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2_KHR,
VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2,
// Provided by VK_KHR_get_memory_requirements2
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2_KHR =
VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2,
// Provided by VK_KHR_get_memory_requirements2
VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2_KHR =
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO,
// Provided by VK_KHR_image_format_list
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHR =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO_EXT =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO_EXT =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES_EXT =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES_EXT =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO_EXT =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO_EXT =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT_EXT =
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT_EXT =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES_KHR =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES,
 // Provided by VK_KHR_maintenance3
 VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT_KHR =
 VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT,
 // Provided by VK_KHR_shader_subgroup_extended_types
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES,
 // Provided by VK_KHR_8bit_storage
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES,
 // Provided by VK_KHR_shader_atomic_int64
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES,
 // Provided by VK_KHR_driver_properties
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES,
 // Provided by VK_KHR_shader_float_controls
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES,
 // Provided by VK_KHR_depth_stencil_resolve
 VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE_KHR =
 VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE,
 // Provided by VK_KHR_timeline_semaphore
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES,
 // Provided by VK_KHR_timeline_semaphore
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES,
 // Provided by VK_KHR_timeline_semaphore
 VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO_KHR = VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO,
 // Provided by VK_KHR_timeline_semaphore
 VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO_KHR =
 VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO_KHR = VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO,
 // Provided by VK_KHR_timeline_semaphore
 VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO_KHR =
 VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO_INTEL =
 VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_QUERY_CREATE_INFO_INTEL,
 // Provided by VK_KHR_vulkan_memory_model
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES_KHR =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES,
 // Provided by VK_EXT_scalar_block_layout
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES_EXT =
 VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES,
Each value corresponds to a particular structure with a sType member with a matching name. As a general rule, the name of each VkStructureType value is obtained by taking the name of the structure, stripping the leading Vk, prefixing each capital letter with _, converting the entire resulting string to upper case, and prefixing it with VK_STRUCTURE_TYPE_. For example, structures of type VkImageCreateInfo correspond to a VkStructureType of VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO, and thus its sType member must equal that when it is passed to the API.

The values VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO and VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO are reserved for internal use by the loader, and do not have corresponding Vulkan structures in this Specification.
Valid Usage for Structure Pointer Chains

Any parameter that is a structure containing a `void* pNext` member must have a value of `pNext` that is either `NULL`, or is a pointer to a valid extending structure, containing `sType` and `pNext` members as described in the Vulkan Documentation and Extensions document in the section “Extension Interactions”. The set of structures connected by `pNext` pointers is referred to as a `pNext chain`.

Each structure present in the `pNext` chain must be defined at runtime by either:

- a core version which is supported
- an extension which is enabled
- a supported device extension in the case of physical-device-level functionality added by the device extension

Each type of extending structure must not appear more than once in a `pNext` chain, including any aliases. This general rule may be explicitly overridden for specific structures.

Any component of the implementation (the loader, any enabled layers, and drivers) must skip over, without processing (other than reading the `sType` and `pNext` members) any extending structures in the chain not defined by core versions or extensions supported by that component.

As a convenience to implementations and layers needing to iterate through a structure pointer chain, the Vulkan API provides two base structures. These structures allow for some type safety, and can be used by Vulkan API functions that operate on generic inputs and outputs.

The `VkBaseInStructure` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBaseInStructure {
    VkStructureType                    sType;
    const struct VkBaseInStructure*    pNext;
} VkBaseInStructure;
```

- `sType` is the structure type of the structure being iterated through.
- `pNext` is `NULL` or a pointer to the next structure in a structure chain.

`VkBaseInStructure` can be used to facilitate iterating through a read-only structure pointer chain.

The `VkBaseOutStructure` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBaseOutStructure {
    VkStructureType               sType;
    struct VkBaseOutStructure*    pNext;
} VkBaseOutStructure;
```

- `sType` is the structure type of the structure being iterated through.
• `pNext` is `NULL` or a pointer to the next structure in a structure chain.

`VkBaseOutStructure` can be used to facilitate iterating through a structure pointer chain that returns data back to the application.

### Valid Usage for Nested Structures

The above conditions also apply recursively to members of structures provided as input to a command, either as a direct argument to the command, or themselves a member of another structure.

Specifics on valid usage of each command are covered in their individual sections.

### Valid Usage for Extensions

Instance-level functionality or behavior added by an instance extension to the API must not be used unless that extension is supported by the instance as determined by `vkEnumerateInstanceExtensionProperties`, and that extension is enabled in `VkInstanceCreateInfo`.

Physical-device-level functionality or behavior added by an instance extension to the API must not be used unless that extension is supported by the instance as determined by `vkEnumerateInstanceExtensionProperties`, and that extension is enabled in `VkInstanceCreateInfo`.

Physical-device-level functionality or behavior added by a device extension to the API must not be used unless the conditions described in Extending Physical Device Core Functionality are met.

Device functionality or behavior added by a device extension to the API must not be used unless that extension is supported by the device as determined by `vkEnumerateDeviceExtensionProperties`, and that extension is enabled in `VkDeviceCreateInfo`.

### Valid Usage for Newer Core Versions

Instance-level functionality or behavior added by a new core version of the API must not be used unless it is supported by the instance as determined by `vkEnumerateInstanceVersion` and the specified version of `VkApplicationInfo::apiVersion`.

Physical-device-level functionality or behavior added by a new core version of the API must not be used unless it is supported by the physical device as determined by `VkPhysicalDeviceProperties::apiVersion` and the specified version of `VkApplicationInfo::apiVersion`.

Device-level functionality or behavior added by a new core version of the API must not be used unless it is supported by the device as determined by `VkPhysicalDeviceProperties::apiVersion` and the specified version of `VkApplicationInfo::apiVersion`.

### 3.7.3. Return Codes

While the core Vulkan API is not designed to capture incorrect usage, some circumstances still require return codes. Commands in Vulkan return their status via return codes that are in one of two categories:

• Successful completion codes are returned when a command needs to communicate success or
status information. All successful completion codes are non-negative values.

- Run time error codes are returned when a command needs to communicate a failure that could only be detected at runtime. All runtime error codes are negative values.

All return codes in Vulkan are reported via `VkResult` return values. The possible codes are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkResult {
    VK_SUCCESS = 0,
    VK_NOT_READY = 1,
    VK_TIMEOUT = 2,
    VK_EVENT_SET = 3,
    VK_EVENT_RESET = 4,
    VK_INCOMPLETE = 5,
    VK_ERROR_OUT_OF_HOST_MEMORY = -1,
    VK_ERROR_OUT_OF_DEVICE_MEMORY = -2,
    VK_ERROR_INITIALIZATION_FAILED = -3,
    VK_ERROR_DEVICE_LOST = -4,
    VK_ERROR_MEMORY_MAP_FAILED = -5,
    VK_ERROR_LAYER_NOT_PRESENT = -6,
    VK_ERROR_EXTENSION_NOT_PRESENT = -7,
    VK_ERROR_FEATURE_NOT_PRESENT = -8,
    VK_ERROR_INCOMPATIBLE_DRIVER = -9,
    VK_ERROR_TOO_MANY_OBJECTS = -10,
    VK_ERROR_FORMAT_NOT_SUPPORTED = -11,
    VK_ERROR_FRAGMENTED_POOL = -12,
    VK_ERROR_UNKNOWN = -13,
    // Provided by VK_VERSION_1_1
    VK_ERROR_OUT_OF_POOL_MEMORY = -1000069000,
    // Provided by VK_VERSION_1_1
    VK_ERROR_INVALID_EXTERNAL_HANDLE = -1000072003,
    VK_ERROR_FRAGMENTATION = -1000161000,
    VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS = -1000257000,
    // Provided by VK_KHR_surface
    VK_ERROR_SURFACE_LOST_KHR = -1000000000,
    // Provided by VK_KHR_surface
    VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000000001,
    // Provided by VK_KHR_swapchain
    VK_SUBOPTIMAL_KHR = 1000001003,
    // Provided by VK_KHR_swapchain
    VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
    // Provided by VK_KHR_display_swapchain
    VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
    // Provided by VK_EXT_debug_report
    VK_ERROR_VALIDATION_FAILED_EXT = -1000011001,
    // Provided by VK_NV_glsl_shader
    VK_ERROR_INVALID_SHADER_NV = -1000012000,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT = -1000158000,
    // Provided by VK_EXT_global_priority
};
```
VK_ERROR_NOT_PERMITTED_EXT = -1000174001,
// Provided by VK_EXT_full_screen_exclusive
VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT = -1000255000,
// Provided by VK_KHR_deferred_host_operations
VK_THREAD_IDLE_KHR = 1000268000,
// Provided by VK_KHR_deferred_host_operations
VK_THREAD_DONE_KHR = 1000268001,
// Provided by VK_KHR_deferred_host_operations
VK_OPERATION_DEFERRED_KHR = 1000268002,
// Provided by VK_KHR_deferred_host_operations
VK_OPERATION_NOT_DEFERRED_KHR = 1000268003,
// Provided by VK_KHR_maintenance1
VK_PIPELINE_COMPILE_REQUIRED_EXT = 1000297000,
// Provided by VK_KHR_maintenance1
VK_ERROR_OUT_OF_POOL_MEMORY_KHR = VK_ERROR_OUT_OF_POOL_MEMORY,
// Provided by VK_KHR_external_memory
VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR = VK_ERROR_INVALID_EXTERNAL_HANDLE,
// Provided by VK_EXT_descriptor_indexing
VK_ERROR_FRAGMENTATION_EXT = VK_ERROR_FRAGMENTATION,
// Provided by VK_EXT_buffer_device_address
VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR = VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR,
VK_ERROR_PIPELINE_COMPILE_REQUIRED_EXT = VK_PIPELINE_COMPILE_REQUIRED_EXT,
} VkResult;

Success Codes
- **VK_SUCCESS** Command successfully completed
- **VK_NOT_READY** A fence or query has not yet completed
- **VK_TIMEOUT** A wait operation has not completed in the specified time
- **VK_EVENT_SET** An event is signaled
- **VK_EVENT_RESET** An event is unsignaled
- **VK_INCOMPLETE** A return array was too small for the result
- **VK_SUBOPTIMAL_KHR** A swapchain no longer matches the surface properties exactly, but can still be used to present to the surface successfully.
- **VK_THREAD_IDLE_KHR** A deferred operation is not complete but there is currently no work for this thread to do at the time of this call.
- **VK_THREAD_DONE_KHR** A deferred operation is not complete but there is no work remaining to assign to additional threads.
- **VK_OPERATION_DEFERRED_KHR** A deferred operation was requested and at least some of the work was deferred.
- **VK_OPERATION_NOT_DEFERRED_KHR** A deferred operation was requested and no operations were deferred.
- **VK_PIPELINE_COMPILE_REQUIRED_EXT** A requested pipeline creation would have required
compilation, but the application requested compilation to not be performed.

**Error codes**

- **VK_ERROR_OUT_OF_HOST_MEMORY** A host memory allocation has failed.
- **VK_ERROR_OUT_OF_DEVICE_MEMORY** A device memory allocation has failed.
- **VK_ERROR_INITIALIZATION_FAILED** Initialization of an object could not be completed for implementation-specific reasons.
- **VK_ERROR_DEVICE_LOST** The logical or physical device has been lost. See Lost Device
- **VK_ERROR_MEMORY_MAP_FAILED** Mapping of a memory object has failed.
- **VK_ERROR_LAYER_NOT_PRESENT** A requested layer is not present or could not be loaded.
- **VK_ERROR_EXTENSION_NOT_PRESENT** A requested extension is not supported.
- **VK_ERROR_FEATURE_NOT_PRESENT** A requested feature is not supported.
- **VK_ERROR_INCOMPATIBLE_DRIVER** The requested version of Vulkan is not supported by the driver or is otherwise incompatible for implementation-specific reasons.
- **VK_ERROR_TOO_MANY_OBJECTS** Too many objects of the type have already been created.
- **VK_ERROR_FORMAT_NOT_SUPPORTED** A requested format is not supported on this device.
- **VK_ERROR_FRAGMENTED_POOL** A pool allocation has failed due to fragmentation of the pool’s memory. This must only be returned if no attempt to allocate host or device memory was made to accommodate the new allocation. This should be returned in preference to **VK_ERROR_OUT_OF_POOL_MEMORY**, but only if the implementation is certain that the pool allocation failure was due to fragmentation.
- **VK_ERROR_SURFACE_LOST_KHR** A surface is no longer available.
- **VK_ERROR_NATIVE_WINDOW_IN_USE_KHR** The requested window is already in use by Vulkan or another API in a manner which prevents it from being used again.
- **VK_ERROR_OUT_OF_DATE_KHR** A surface has changed in such a way that it is no longer compatible with the swapchain, and further presentation requests using the swapchain will fail. Applications must query the new surface properties and recreate their swapchain if they wish to continue presenting to the surface.
- **VK_ERROR_INCOMPATIBLE_DISPLAY_KHR** The display used by a swapchain does not use the same presentable image layout, or is incompatible in a way that prevents sharing an image.
- **VK_ERROR_INVALID_SHADER_NV** One or more shaders failed to compile or link. More details are reported back to the application via **VK_EXT_debug_report** if enabled.
- **VK_ERROR_OUT_OF_POOL_MEMORY** A pool memory allocation has failed. This must only be returned if no attempt to allocate host or device memory was made to accommodate the new allocation. If the failure was definitely due to fragmentation of the pool, **VK_ERROR_FRAGMENTED_POOL** should be returned instead.
- **VK_ERROR_INVALID_EXTERNAL_HANDLE** An external handle is not a valid handle of the specified type.
- **VK_ERROR_FRAGMENTATION** A descriptor pool creation has failed due to fragmentation.
- **VK_ERROR_INVALID_DEVICE_ADDRESS_EXT** A buffer creation failed because the requested address is not available.
• **VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS** A buffer creation or memory allocation failed because the requested address is not available. A shader group handle assignment failed because the requested shader group handle information is no longer valid.

• **VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT** An operation on a swapchain created with `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT` failed as it did not have exclusive full-screen access. This may occur due to implementation-dependent reasons, outside of the application’s control.

• **VK_ERROR_UNKNOWN** An unknown error has occurred; either the application has provided invalid input, or an implementation failure has occurred.

If a command returns a runtime error, unless otherwise specified any output parameters will have undefined contents, except that if the output parameter is a structure with `sType` and `pNext` fields, those fields will be unmodified. Any structures chained from `pNext` will also have undefined contents, except that `sType` and `pNext` will be unmodified.

**VK_ERROR_OUT_OF_*_MEMORY** errors do not modify any currently existing Vulkan objects. Objects that have already been successfully created can still be used by the application.

---

**Note**

As a general rule, `Free`, `Release`, and `Reset` commands do not return `VK_ERROR_OUT_OF_HOST_MEMORY`, while any other command with a return code may return it. Any exceptions from this rule are described for those commands.

**VK_ERROR_UNKNOWN** will be returned by an implementation when an unexpected error occurs that cannot be attributed to valid behavior of the application and implementation. Under these conditions, it may be returned from any command returning a `VkResult`.

---

**Note**

`VK_ERROR_UNKNOWN` is not expected to ever be returned if the application behavior is valid, and if the implementation is bug-free. If `VK_ERROR_UNKNOWN` is received, the application should be checked against the latest validation layers to verify correct behavior as much as possible. If no issues are identified it could be an implementation issue, and the implementor should be contacted for support.

Performance-critical commands generally do not have return codes. If a runtime error occurs in such commands, the implementation will defer reporting the error until a specified point. For commands that record into command buffers (`vkCmd*`) runtime errors are reported by `vkEndCommandBuffer`.

### 3.8. Numeric Representation and Computation

Implementations normally perform computations in floating-point, and must meet the range and precision requirements defined under “Floating-Point Computation” below.

These requirements only apply to computations performed in Vulkan operations outside of shader execution, such as texture image specification and sampling, and per-fragment operations. Range and precision requirements during shader execution differ and are specified by the *Precision and...*
Operation of SPIR-V Instructions section.

In some cases, the representation and/or precision of operations is implicitly limited by the specified format of vertex or texel data consumed by Vulkan. Specific floating-point formats are described later in this section.

### 3.8.1. Floating-Point Computation

Most floating-point computation is performed in SPIR-V shader modules. The properties of computation within shaders are constrained as defined by the Precision and Operation of SPIR-V Instructions section.

Some floating-point computation is performed outside of shaders, such as viewport and depth range calculations. For these computations, we do not specify how floating-point numbers are to be represented, or the details of how operations on them are performed, but only place minimal requirements on representation and precision as described in the remainder of this section.

We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in $10^5$. The maximum representable magnitude for all floating-point values must be at least $2^{32}$.

$x \times 0 = 0 \times x = 0$ for any non-infinite and non-NaN $x$.

$1 \times x = x \times 1 = x$.

$x + 0 = 0 + x = x$.

$0^0 = 1$.

Occasionally, further requirements will be specified. Most single-precision floating-point formats meet these requirements.

The special values Inf and -Inf encode values with magnitudes too large to be represented; the special value NaN encodes “Not A Number” values resulting from undefined arithmetic operations such as $0 / 0$. Implementations may support Inf and NaN in their floating-point computations.

### 3.8.2. Floating-Point Format Conversions

When a value is converted to a defined floating-point representation, finite values falling between two representable finite values are rounded to one or the other. The rounding mode is not defined. Finite values whose magnitude is larger than that of any representable finite value may be rounded either to the closest representable finite value or to the appropriately signed infinity. For unsigned destination formats any negative values are converted to zero. Positive infinity is converted to positive infinity; negative infinity is converted to negative infinity in signed formats and to zero in unsigned formats; and any NaN is converted to a NaN.
3.8.3. 16-Bit Floating-Point Numbers

16-bit floating point numbers are defined in the “16-bit floating point numbers” section of the Khronos Data Format Specification.

3.8.4. Unsigned 11-Bit Floating-Point Numbers

Unsigned 11-bit floating point numbers are defined in the “Unsigned 11-bit floating point numbers” section of the Khronos Data Format Specification.

3.8.5. Unsigned 10-Bit Floating-Point Numbers

Unsigned 10-bit floating point numbers are defined in the “Unsigned 10-bit floating point numbers” section of the Khronos Data Format Specification.

3.8.6. General Requirements

Any representable floating-point value in the appropriate format is legal as input to a Vulkan command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to Vulkan interruption or termination. For example, providing a negative zero (where applicable) or a denormalized number to a Vulkan command must yield deterministic results, while providing a NaN or Inf yields unspecified results.

Some calculations require division. In such cases (including implied divisions performed by vector normalization), division by zero produces an unspecified result but must not lead to Vulkan interruption or termination.

3.9. Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth components are represented as integers, they are often (but not always) considered to be normalized. Normalized integer values are treated specially when being converted to and from floating-point values, and are usually referred to as normalized fixed-point.

In the remainder of this section, b denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined by the API, b is the bit width of that type. When the integer comes from an image containing color or depth component texels, b is the number of bits allocated to that component in its specified image format.

The signed and unsigned fixed-point representations are assumed to be b-bit binary two’s-complement integers and binary unsigned integers, respectively.

3.9.1. Conversion from Normalized Fixed-Point to Floating-Point

Unsigned normalized fixed-point integers represent numbers in the range [0,1]. The conversion from an unsigned normalized fixed-point value c to the corresponding floating-point value f is defined as
Signed normalized fixed-point integers represent numbers in the range \([-1,1]\). The conversion from a signed normalized fixed-point value \(c\) to the corresponding floating-point value \(f\) is performed using

\[
f = \frac{c}{2^b - 1}
\]

Only the range \([-2^{b-1} + 1, 2^{b-1} - 1]\) is used to represent signed fixed-point values in the range \([-1,1]\). For example, if \(b = 8\), then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point.

Note that while zero is exactly expressible in this representation, one value (-128 in the example) is outside the representable range, and implementations must clamp it to -1.0. Where the value is subject to further processing by the implementation, eg. during texture filtering, values less than -1.0 may be used but the result must be clamped before the value is returned to shaders.

### 3.9.2. Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value \(f\) to the corresponding unsigned normalized fixed-point value \(c\) is defined by first clamping \(f\) to the range \([0,1]\), then computing

\[
c = \text{convertFloatToUint}(f \times (2^b - 1), b)
\]

where \text{convertFloatToUint}(r,b) returns one of the two unsigned binary integer values with exactly \(b\) bits which are closest to the floating-point value \(r\). Implementations should round to nearest. If \(r\) is equal to an integer, then that integer value must be returned. In particular, if \(f\) is equal to 0.0 or 1.0, then \(c\) must be assigned 0 or \(2^b - 1\), respectively.

The conversion from a floating-point value \(f\) to the corresponding signed normalized fixed-point value \(c\) is performed by clamping \(f\) to the range \([-1,1]\), then computing

\[
c = \text{convertFloatToInt}(f \times (2^{b-1} - 1), b)
\]

where \text{convertFloatToInt}(r,b) returns one of the two signed two’s-complement binary integer values with exactly \(b\) bits which are closest to the floating-point value \(r\). Implementations should round to nearest. If \(r\) is equal to an integer, then that integer value must be returned. In particular, if \(f\) is equal to -1.0, 0.0, or 1.0, then \(c\) must be assigned \(-2^{b-1}\), 0, or \(2^{b-1} - 1\), respectively.

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point.

### 3.10. Common Object Types

Some types of Vulkan objects are used in many different structures and command parameters, and are described here. These types include offsets, extents, and rectangles.
### 3.10.1. Offsets

Offsets are used to describe a pixel location within an image or framebuffer, as an (x,y) location for two-dimensional images, or an (x,y,z) location for three-dimensional images.

A two-dimensional offset is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkOffset2D {
    int32_t    x;
    int32_t    y;
} VkOffset2D;
```

- x is the x offset.
- y is the y offset.

A three-dimensional offset is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkOffset3D {
    int32_t    x;
    int32_t    y;
    int32_t    z;
} VkOffset3D;
```

- x is the x offset.
- y is the y offset.
- z is the z offset.

### 3.10.2. Extents

Extents are used to describe the size of a rectangular region of pixels within an image or framebuffer, as (width,height) for two-dimensional images, or as (width,height,depth) for three-dimensional images.

A two-dimensional extent is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtent2D {
    uint32_t    width;
    uint32_t    height;
} VkExtent2D;
```

- width is the width of the extent.
- height is the height of the extent.
A three-dimensional extent is defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtent3D {
    uint32_t    width;
    uint32_t    height;
    uint32_t    depth;
} VkExtent3D;
```

- `width` is the width of the extent.
- `height` is the height of the extent.
- `depth` is the depth of the extent.

### 3.10.3. Rectangles

Rectangles are used to describe a specified rectangular region of pixels within an image or framebuffer. Rectangles include both an offset and an extent of the same dimensionality, as described above. Two-dimensional rectangles are defined by the structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkRect2D {
    VkOffset2D    offset;
    VkExtent2D    extent;
} VkRect2D;
```

- `offset` is a `VkOffset2D` specifying the rectangle offset.
- `extent` is a `VkExtent2D` specifying the rectangle extent.
Chapter 4. Initialization

Before using Vulkan, an application must initialize it by loading the Vulkan commands, and creating a `VkInstance` object.

### 4.1. Command Function Pointers

Vulkan commands are not necessarily exposed by static linking on a platform. Commands to query function pointers for Vulkan commands are described below.

**Note**

When extensions are promoted or otherwise incorporated into another extension or Vulkan core version, command aliases may be included. Whilst the behavior of each command alias is identical, the behavior of retrieving each alias's function pointer is not. A function pointer for a given alias can only be retrieved if the extension or version that introduced that alias is supported and enabled, irrespective of whether any other alias is available.

Function pointers for all Vulkan commands can be obtained with the command:

```c
// Provided by VK_VERSION_1_0
PFN_vkVoidFunction vkGetInstanceProcAddr(
    VkInstance instance,
    const char* pName);
```

- `instance` is the instance that the function pointer will be compatible with, or `NULL` for commands not dependent on any instance.
- `pName` is the name of the command to obtain.

`vkGetInstanceProcAddr` itself is obtained in a platform- and loader- specific manner. Typically, the loader library will export this command as a function symbol, so applications can link against the loader library, or load it dynamically and look up the symbol using platform-specific APIs.

The table below defines the various use cases for `vkGetInstanceProcAddr` and expected return value (“fp” is “function pointer”) for each case.

The returned function pointer is of type `PFN_vkVoidFunction`, and must be cast to the type of the command being queried before use.

**Table 1. `vkGetInstanceProcAddr` behavior**

<table>
<thead>
<tr>
<th>instance</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>*1 NULL</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid non-NULL</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>instance</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>NULL</td>
<td>vkEnumerateInstanceVersion</td>
<td>fp</td>
</tr>
</tbody>
</table>
### Table

<table>
<thead>
<tr>
<th>instance</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>vkEnumerateInstanceExtensionProperties</td>
<td>fp</td>
</tr>
<tr>
<td>NULL</td>
<td>vkEnumerateInstanceLayerProperties</td>
<td>fp</td>
</tr>
<tr>
<td>NULL</td>
<td>vkCreateInstance</td>
<td>fp</td>
</tr>
<tr>
<td>instance</td>
<td>core Vulkan command</td>
<td>fp(^2)</td>
</tr>
<tr>
<td>instance</td>
<td>enabled instance extension commands for instance</td>
<td>fp(^2)</td>
</tr>
<tr>
<td>instance</td>
<td>available device extension(^3) commands for instance</td>
<td>fp(^2)</td>
</tr>
<tr>
<td>any other case, not covered above</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

1

"*" means any representable value for the parameter (including valid values, invalid values, and NULL).

2

The returned function pointer **must** only be called with a dispatchable object (the first parameter) that is `instance` or a child of `instance`, e.g. `VkInstance`, `VkPhysicalDevice`, `VkDevice`, `VkQueue`, or `VkCommandBuffer`.

3

An “available device extension” is a device extension supported by any physical device enumerated by `instance`.

### Valid Usage (Implicit)

- **VUID-vkGetInstanceProcAddr-instance-parameter**
  If `instance` is not NULL, `instance` must be a valid `VkInstance` handle

- **VUID-vkGetInstanceProcAddr-pName-parameter**
  `pName` must be a null-terminated UTF-8 string

In order to support systems with multiple Vulkan implementations, the function pointers returned by `vkGetInstanceProcAddr` may point to dispatch code that calls a different real implementation for different `VkDevice` objects or their child objects. The overhead of the internal dispatch for `VkDevice` objects can be avoided by obtaining device-specific function pointers for any commands that use a device or device-child object as their dispatchable object. Such function pointers **can** be obtained with the command:
The table below defines the various use cases for `vkGetDeviceProcAddr` and expected return value for each case.

The returned function pointer is of type `PFN_vkVoidFunction`, and **must** be cast to the type of the command being queried before use. The function pointer **must** only be called with a dispatchable object (the first parameter) that is `device` or a child of `device`.

<table>
<thead>
<tr>
<th>device</th>
<th>pName</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NULL</code></td>
<td><code>*</code>1</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid device</td>
<td><code>*</code>1</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td><code>NULL</code></td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>core device-level Vulkan command²</td>
<td>fp³</td>
</tr>
<tr>
<td>device</td>
<td>enabled extension device-level commands²</td>
<td>fp³</td>
</tr>
<tr>
<td>any other case, not covered above</td>
<td><code>NULL</code></td>
<td></td>
</tr>
</tbody>
</table>

1

"*" means any representable value for the parameter (including valid values, invalid values, and `NULL`).

2

In this function, device-level excludes all physical-device-level commands.

3

The returned function pointer **must** only be called with a dispatchable object (the first parameter) that is `device` or a child of `device` e.g. `VkDevice`, `VkQueue`, or `VkCommandBuffer`.

---

**Valid Usage (Implicit)**

- `VUID-vkGetDeviceProcAddr-device-parameter`
  - `device` **must** be a valid `VkDevice` handle

- `VUID-vkGetDeviceProcAddr-pName-parameter`
  - `pName` **must** be a null-terminated UTF-8 string

The definition of `PFN_vkVoidFunction` is:
4.1.1. Extending Physical Device Core Functionality

New core physical-device-level functionality can be used when the physical-device version is greater than or equal to the version of Vulkan that added the new functionality. The Vulkan version supported by a physical device can be obtained by calling `vkGetPhysicalDeviceProperties`.

4.1.2. Extending Physical Device From Device Extensions

When the `VK_KHR_get_physical_device_properties2` extension is enabled, or when both the instance and the physical-device versions are at least 1.1, physical-device-level functionality of a device extension can be used with a physical device if the corresponding extension is enumerated by `vkEnumerateDeviceExtensionProperties` for that physical device, even before a logical device has been created.

To obtain a function pointer for a physical-device-level command from a device extension, an application can use `vkGetInstanceProcAddr`. This function pointer may point to dispatch code, which calls a different real implementation for different `VkPhysicalDevice` objects. Applications must not use a `VkPhysicalDevice` in any command added by an extension or core version that is not supported by that physical device.

Device extensions may define structures that can be added to the `pNext` chain of physical-device-level commands.

4.2. Instances

There is no global state in Vulkan and all per-application state is stored in a `VkInstance` object. Creating a `VkInstance` object initializes the Vulkan library and allows the application to pass information about itself to the implementation.

Instances are represented by `VkInstance` handles:

```c
// Provided by VK_VERSION_1_0
typedef void (VKAPI_PTR *PFN_vkVoidFunction)(void);
```

To query the version of instance-level functionality supported by the implementation, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkEnumerateInstanceVersion(
    uint32_t* pApiVersion);
```

- `pApiVersion` is a pointer to a `uint32_t`, which is the version of Vulkan supported by instance-level functionality, encoded as described in Version Numbers.
Note
The intended behaviour of `vkEnumerateInstanceVersion` is that an implementation should not need to perform memory allocations and should unconditionally return `VK_SUCCESS`. The loader, and any enabled layers, may return `VK_ERROR_OUT_OF_HOST_MEMORY` in the case of a failed memory allocation.

Valid Usage (Implicit)

- VUID-vkEnumerateInstanceVersion-pApiVersion-parameter
  
  `pApiVersion` must be a valid pointer to a `uint32_t` value

Return Codes

Success
- `VK_SUCCESS`

Failure
- `VK_ERROR_OUT_OF_HOST_MEMORY`

To create an instance object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkInstance* pInstance);
```

- `pCreateInfo` is a pointer to a `VkInstanceCreateInfo` structure controlling creation of the instance.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pInstance` points a `VkInstance` handle in which the resulting instance is returned.

`vkCreateInstance` verifies that the requested layers exist. If not, `vkCreateInstance` will return `VK_ERROR_LAYER_NOT_PRESENT`. Next `vkCreateInstance` verifies that the requested extensions are supported (e.g. in the implementation or in any enabled instance layer) and if any requested extension is not supported, `vkCreateInstance` must return `VK_ERROR_EXTENSION_NOT_PRESENT`. After verifying and enabling the instance layers and extensions the `VkInstance` object is created and returned to the application. If a requested extension is only supported by a layer, both the layer and the extension need to be specified at `vkCreateInstance` time for the creation to succeed.
Valid Usage

- VUID-vkCreateInstance-ppEnabledExtensionNames-01388
  All required extensions for each extension in the VkInstanceCreateInfo::ppEnabledExtensionNames list must also be present in that list

Valid Usage (Implicit)

- VUID-vkCreateInstance-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkInstanceCreateInfo structure

- VUID-vkCreateInstance-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateInstance-pInstance-parameter
  pInstance must be a valid pointer to a VkInstance handle

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED
- VK_ERROR_LAYER_NOT_PRESENT
- VK_ERROR_EXTENSION_NOT_PRESENT
- VK_ERROR_INCOMPATIBLE_DRIVER

The VkInstanceCreateInfo structure is defined as:
// Provided by VK_VERSION_1_0

typedef struct VkInstanceCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkInstanceCreateFlags       flags;
    const VkApplicationInfo*    pApplicationInfo;
    uint32_t                    enabledLayerCount;
    const char* const*          ppEnabledLayerNames;
    uint32_t                    enabledExtensionCount;
    const char* const*          ppEnabledExtensionNames;
} VkInstanceCreateInfo;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **flags** is reserved for future use.

• **pApplicationInfo** is NULL or a pointer to a VkApplicationInfo structure. If not NULL, this information helps implementations recognize behavior inherent to classes of applications. VkApplicationInfo is defined in detail below.

• **enabledLayerCount** is the number of global layers to enable.

• **ppEnabledLayerNames** is a pointer to an array of enabledLayerCount null-terminated UTF-8 strings containing the names of layers to enable for the created instance. The layers are loaded in the order they are listed in this array, with the first array element being the closest to the application, and the last array element being the closest to the driver. See the Layers section for further details.

• **enabledExtensionCount** is the number of global extensions to enable.

• **ppEnabledExtensionNames** is a pointer to an array of enabledExtensionCount null-terminated UTF-8 strings containing the names of extensions to enable.
Valid Usage (Implicit)

- **VUID-VkInstanceCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO`

- **VUID-VkInstanceCreateInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDebugReportCallbackCreateInfoEXT`, `VkDebugUtilsMessengerCreateInfoEXT`, `VkValidationFeaturesEXT`, or `VkValidationFlagsEXT`

- **VUID-VkInstanceCreateInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique, with the exception of structures of type `VkDebugUtilsMessengerCreateInfoEXT`

- **VUID-VkInstanceCreateInfo-flags-zerobitmask**
  
  `flags` must be 0

- **VUID-VkInstanceCreateInfo-pApplicationInfo-parameter**
  
  If `pApplicationInfo` is not `NULL`, `pApplicationInfo` must be a valid pointer to a valid `VkApplicationInfo` structure

- **VUID-VkInstanceCreateInfo-ppEnabledLayerNames-parameter**
  
  If `enabledLayerCount` is not 0, `ppEnabledLayerNames` must be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings

- **VUID-VkInstanceCreateInfo-ppEnabledExtensionNames-parameter**
  
  If `enabledExtensionCount` is not 0, `ppEnabledExtensionNames` must be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings

---

`// Provided by VK_VERSION_1_0`

typedef VkFlags VkInstanceCreateFlags;

`VkInstanceCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

When creating a Vulkan instance for which you wish to disable validation checks, add a `VkValidationFlagsEXT` structure to the `pNext` chain of the ` VkInstanceCreateInfo` structure, specifying the checks to be disabled.

---

`// Provided by VK_EXT_validation_flags`

typedef struct VkValidationFlagsEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t disabledValidationCheckCount;
    const VkValidationCheckEXT* pDisabledValidationChecks;
} VkValidationFlagsEXT;

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to a structure extending this structure.
• `disabledValidationCheckCount` is the number of checks to disable.

• `pDisabledValidationChecks` is a pointer to an array of `VkValidationCheckEXT` values specifying the validation checks to be disabled.

---

**Valid Usage (Implicit)**

- **VUID-VkValidationFlagsEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT`

- **VUID-VkValidationFlagsEXT-pDisabledValidationChecks-parameter**
  
  `pDisabledValidationChecks` must be a valid pointer to an array of `disabledValidationCheckCount` valid `VkValidationCheckEXT` values

- **VUID-VkValidationFlagsEXT-disabledValidationCheckCount-arraylength**
  
  `disabledValidationCheckCount` must be greater than 0

Possible values of elements of the `VkValidationFlagsEXT::pDisabledValidationChecks` array, specifying validation checks to be disabled, are:

```c
// Provided by VK_EXT_validation_flags
typedef enum VkValidationCheckEXT {
    VK_VALIDATION_CHECK_ALL_EXT = 0,
    VK_VALIDATION_CHECK_SHADERS_EXT = 1,
} VkValidationCheckEXT;
```

- `VK_VALIDATION_CHECK_ALL_EXT` specifies that all validation checks are disabled.
- `VK_VALIDATION_CHECK_SHADERS_EXT` specifies that shader validation is disabled.

When creating a Vulkan instance for which you wish to enable or disable specific validation features, add a `VkValidationFeaturesEXT` structure to the `pNext` chain of the `VkInstanceCreateInfo` structure, specifying the features to be enabled or disabled.

```c
// Provided by VK_EXT_validation_features
typedef struct VkValidationFeaturesEXT {
    VkStructureType                         sType;
    const void*                             pNext;
    uint32_t                                enabledValidationFeatureCount;
    const VkValidationFeatureEnableEXT*     pEnabledValidationFeatures;
    uint32_t                                disabledValidationFeatureCount;
    const VkValidationFeatureDisableEXT*    pDisabledValidationFeatures;
} VkValidationFeaturesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `enabledValidationFeatureCount` is the number of features to enable.
• **pEnabledValidationFeatures** is a pointer to an array of `VkValidationFeatureEnableEXT` values specifying the validation features to be enabled.

• **disabledValidationFeatureCount** is the number of features to disable.

• **pDisabledValidationFeatures** is a pointer to an array of `VkValidationFeatureDisableEXT` values specifying the validation features to be disabled.

---

### Valid Usage

• VUID-VkValidationFeaturesEXT-pEnabledValidationFeatures-02967
  If the `pEnabledValidationFeatures` array contains `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT`, then it must also contain `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT`.

• VUID-VkValidationFeaturesEXT-pEnabledValidationFeatures-02968
  If the `pEnabledValidationFeatures` array contains `VK_VALIDATION_FEATURE_ENABLE_DEBUG_PRINTF_EXT`, then it must not contain `VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT`.

---

### Valid Usage (Implicit)

• VUID-VkValidationFeaturesEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT`.

• VUID-VkValidationFeaturesEXT-pEnabledValidationFeatures-parameter
  If `enabledValidationFeatureCount` is not 0, `pEnabledValidationFeatures` must be a valid pointer to an array of `enabledValidationFeatureCount` valid `VkValidationFeatureEnableEXT` values.

• VUID-VkValidationFeaturesEXT-pDisabledValidationFeatures-parameter
  If `disabledValidationFeatureCount` is not 0, `pDisabledValidationFeatures` must be a valid pointer to an array of `disabledValidationFeatureCount` valid `VkValidationFeatureDisableEXT` values.

---

Possible values of elements of the `VkValidationFeaturesEXT::pEnabledValidationFeatures` array, specifying validation features to be enabled, are:

```c
// Provided by VK_EXT_validation_features
typedef enum VkValidationFeatureEnableEXT {
    VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT = 0,
    VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT = 1,
    VK_VALIDATION_FEATURE_ENABLE_BEST_PRACTICES_EXT = 2,
    VK_VALIDATION_FEATURE_ENABLE_DEBUG_PRINTF_EXT = 3,
    VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT = 4,
} VkValidationFeatureEnableEXT;
```

• **VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_EXT** specifies that GPU-assisted validation is enabled. Activating this feature instruments shader programs to generate additional diagnostic data. This
feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_GPU_ASSISTED_RESERVE_BINDING_SLOT_EXT** specifies that the validation layers reserve a descriptor set binding slot for their own use. The layer reports a value for `VkPhysicalDeviceLimits::maxBoundDescriptorSets` that is one less than the value reported by the device. If the device supports the binding of only one descriptor set, the validation layer does not perform GPU-assisted validation. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_BEST_PRACTICES_EXT** specifies that Vulkan best-practices validation is enabled. Activating this feature enables the output of warnings related to common misuse of the API, but which are not explicitly prohibited by the specification. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_DEBUG_PRINTF_EXT** specifies that the layers will process `debugPrintfEXT` operations in shaders and send the resulting output to the debug callback. This feature is disabled by default.

- **VK_VALIDATION_FEATURE_ENABLE_SYNCHRONIZATION_VALIDATION_EXT** specifies that Vulkan synchronization validation is enabled. This feature reports resource access conflicts due to missing or incorrect synchronization operations between actions (Draw, Copy, Dispatch, Blit) reading or writing the same regions of memory. This feature is disabled by default.

Possible values of elements of the `VkValidationFeaturesEXT::pDisabledValidationFeatures` array, specifying validation features to be disabled, are:

```c
// Provided by VK_EXT_validation_features
typedef enum VkValidationFeatureDisableEXT {
    VK_VALIDATION_FEATURE_DISABLE_ALL_EXT = 0,
    VK_VALIDATION_FEATURE_DISABLE_SHADERS_EXT = 1,
    VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT = 2,
    VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT = 3,
    VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT = 4,
    VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT = 5,
    VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT = 6,
} VkValidationFeatureDisableEXT;
```

- **VK_VALIDATION_FEATURE_DISABLE_ALL_EXT** specifies that all validation checks are disabled.

- **VK_VALIDATION_FEATURE_DISABLE_SHADERS_EXT** specifies that shader validation is disabled. This feature is enabled by default.

- **VK_VALIDATION_FEATURE_DISABLE_THREAD_SAFETY_EXT** specifies that thread safety validation is disabled. This feature is enabled by default.

- **VK_VALIDATION_FEATURE_DISABLE_API_PARAMETERS_EXT** specifies that stateless parameter validation is disabled. This feature is enabled by default.

- **VK_VALIDATION_FEATURE_DISABLE_OBJECT_LIFETIMES_EXT** specifies that object lifetime validation is disabled. This feature is enabled by default.

- **VK_VALIDATION_FEATURE_DISABLE_CORE_CHECKS_EXT** specifies that core validation checks are disabled. This feature is enabled by default. If this feature is disabled, the shader validation and GPU-assisted validation features are also disabled.
• **VK_VALIDATION_FEATURE_DISABLE_UNIQUE_HANDLES_EXT** specifies that protection against duplicate non-dispatchable object handles is disabled. This feature is enabled by default.

**Note**
Disabling checks such as parameter validation and object lifetime validation prevents the reporting of error conditions that can cause other validation checks to behave incorrectly or crash. Some validation checks assume that their inputs are already valid and do not always revalidate them.

**Note**
The **VK_EXT_validation_features** extension subsumes all the functionality provided in the **VK_EXT_validation_flags** extension.

The **VkApplicationInfo** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkApplicationInfo {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pApplicationName;
    uint32_t           applicationVersion;
    const char*        pEngineName;
    uint32_t           engineVersion;
    uint32_t           apiVersion;
} VkApplicationInfo;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **pApplicationName** is **NULL** or is a pointer to a null-terminated UTF-8 string containing the name of the application.
- **applicationVersion** is an unsigned integer variable containing the developer-supplied version number of the application.
- **pEngineName** is **NULL** or is a pointer to a null-terminated UTF-8 string containing the name of the engine (if any) used to create the application.
- **engineVersion** is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application.
- **apiVersion** must be the highest version of Vulkan that the application is designed to use, encoded as described in **Version Numbers**. The patch version number specified in **apiVersion** is ignored when creating an instance object. Only the major and minor versions of the instance **must** match those requested in **apiVersion**.

Vulkan 1.0 implementations were required to return **VK_ERROR_INCOMPATIBLE_DRIVER** if **apiVersion** was larger than 1.0. Implementations that support Vulkan 1.1 or later must not return **VK_ERROR_INCOMPATIBLE_DRIVER** for any value of **apiVersion**.
Note
Because Vulkan 1.0 implementations may fail with VK_ERROR_INCOMPATIBLE_DRIVER, applications should determine the version of Vulkan available before calling vkCreateInstance. If the vkGetInstanceProcAddr returns NULL for vkEnumerateInstanceVersion, it is a Vulkan 1.0 implementation. Otherwise, the application can call vkEnumerateInstanceVersion to determine the version of Vulkan.

As long as the instance supports at least Vulkan 1.1, an application can use different versions of Vulkan with an instance than it does with a device or physical device.

Note
The Khronos validation layers will treat apiVersion as the highest API version the application targets, and will validate API usage against the minimum of that version and the implementation version (instance or device, depending on context). If an application tries to use functionality from a greater version than this, a validation error will be triggered.

For example, if the instance supports Vulkan 1.1 and three physical devices support Vulkan 1.0, Vulkan 1.1, and Vulkan 1.2, respectively, and if the application sets apiVersion to 1.2, the application can use the following versions of Vulkan:

- Vulkan 1.0 can be used with the instance and with all physical devices.
- Vulkan 1.1 can be used with the instance and with the physical devices that support Vulkan 1.1 and Vulkan 1.2.
- Vulkan 1.2 can be used with the physical device that supports Vulkan 1.2.

If we modify the above example so that the application sets apiVersion to 1.1, then the application must not use Vulkan 1.2 functionality on the physical device that supports Vulkan 1.2.

Implicit layers must be disabled if they do not support a version at least as high as apiVersion. See the Vulkan Loader Specification and Architecture Overview document for additional information.

Note
Providing a NULL VkInstanceCreateInfo::pApplicationInfo or providing an apiVersion of 0 is equivalent to providing an apiVersion of VK_MAKE_API_VERSION(0,1,0,0).

Valid Usage

- VUID-VkApplicationInfo-apiVersion-04010
  If apiVersion is not 0, then it must be greater or equal to VK_API_VERSION_1_0
Valid Usage (Implicit)

- VUID-VkApplicationInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_APPLICATION_INFO

- VUID-VkApplicationInfo-pNext-pNext
  pNext must be NULL

- VUID-VkApplicationInfo-pApplicationName-parameter
  If pApplicationName is not NULL, pApplicationName must be a null-terminated UTF-8 string

- VUID-VkApplicationInfo-pEngineName-parameter
  If pEngineName is not NULL, pEngineName must be a null-terminated UTF-8 string

To destroy an instance, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyInstance(
    VkInstance instance,
    const VkAllocationCallbacks* pAllocator);
```

- instance is the handle of the instance to destroy.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyInstance-instance-00629
  All child objects created using instance must have been destroyed prior to destroying instance

- VUID-vkDestroyInstance-instance-00630
  If VkAllocationCallbacks were provided when instance was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyInstance-instance-00631
  If no VkAllocationCallbacks were provided when instance was created, pAllocator must be NULL

Valid Usage (Implicit)

- VUID-vkDestroyInstance-instance-parameter
  If instance is not NULL, instance must be a valid VkInstance handle

- VUID-vkDestroyInstance-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
Host Synchronization

- Host access to instance must be externally synchronized
- Host access to all VkPhysicalDevice objects enumerated from instance must be externally synchronized
Chapter 5. Devices and Queues

Once Vulkan is initialized, devices and queues are the primary objects used to interact with a Vulkan implementation.

Vulkan separates the concept of physical and logical devices. A physical device usually represents a single complete implementation of Vulkan (excluding instance-level functionality) available to the host, of which there are a finite number. A logical device represents an instance of that implementation with its own state and resources independent of other logical devices.

Physical devices are represented by VkPhysicalDevice handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkPhysicalDevice)
```

### 5.1. Physical Devices

To retrieve a list of physical device objects representing the physical devices installed in the system, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumeratePhysicalDevices(
    VkInstance instance,                  
    uint32_t* pPhysicalDeviceCount,      
    VkPhysicalDevice* pPhysicalDevices); 
```

- `instance` is a handle to a Vulkan instance previously created with `vkCreateInstance`.
- `pPhysicalDeviceCount` is a pointer to an integer related to the number of physical devices available or queried, as described below.
- `pPhysicalDevices` is either `NULL` or a pointer to an array of `VkPhysicalDevice` handles.

If `pPhysicalDevices` is `NULL`, then the number of physical devices available is returned in `pPhysicalDeviceCount`. Otherwise, `pPhysicalDeviceCount` must point to a variable set by the user to the number of elements in the `pPhysicalDevices` array, and on return the variable is overwritten with the number of handles actually written to `pPhysicalDevices`. If `pPhysicalDeviceCount` is less than the number of physical devices available, at most `pPhysicalDeviceCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available physical devices were returned.
Valid Usage (Implicit)

- VUID-vkEnumeratePhysicalDevices-instance-parameter
  instance must be a valid VkInstance handle

- VUID-vkEnumeratePhysicalDevices-pPhysicalDeviceCount-parameter
  pPhysicalDeviceCount must be a valid pointer to a uint32_t value

- VUID-vkEnumeratePhysicalDevices-pPhysicalDevices-parameter
  If the value referenced by pPhysicalDeviceCount is not 0, and pPhysicalDevices is not NULL, pPhysicalDevices must be a valid pointer to an array of pPhysicalDeviceCount VkPhysicalDevice handles

Return Codes

Success
- VK_SUCCESS
- VK_INCOMPLETE

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED

To query general properties of physical devices once enumerated, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceProperties(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceProperties*                 pProperties);
```

- physicalDevice is the handle to the physical device whose properties will be queried.
- pProperties is a pointer to a VkPhysicalDeviceProperties structure in which properties are returned.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceProperties-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceProperties-pProperties-parameter
  pProperties must be a valid pointer to a VkPhysicalDeviceProperties structure

The VkPhysicalDeviceProperties structure is defined as:
typedef struct VkPhysicalDeviceProperties {
    uint32_t apiVersion;
    uint32_t driverVersion;
    uint32_t vendorID;
    uint32_t deviceID;
    VkPhysicalDeviceType deviceType;
    char deviceName[VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
    uint8_t pipelineCacheUUID[VK_UUID_SIZE];
    VkPhysicalDeviceLimits limits;
    VkPhysicalDeviceSparseProperties sparseProperties;
} VkPhysicalDeviceProperties;

- **apiVersion** is the version of Vulkan supported by the device, encoded as described in Version Numbers.
- **driverVersion** is the vendor-specified version of the driver.
- **vendorID** is a unique identifier for the vendor (see below) of the physical device.
- **deviceID** is a unique identifier for the physical device among devices available from the vendor.
- **deviceType** is a `VkPhysicalDeviceType` specifying the type of device.
- **deviceName** is an array of `VK_MAX_PHYSICAL_DEVICE_NAME_SIZE` char containing a null-terminated UTF-8 string which is the name of the device.
- **pipelineCacheUUID** is an array of `VK_UUID_SIZE` uint8_t values representing a universally unique identifier for the device.
- **limits** is the `VkPhysicalDeviceLimits` structure specifying device-specific limits of the physical device. See Limits for details.
- **sparseProperties** is the `VkPhysicalDeviceSparseProperties` structure specifying various sparse related properties of the physical device. See Sparse Properties for details.

**Note**
The value of **apiVersion** may be different than the version returned by `vkEnumerateInstanceVersion`; either higher or lower. In such cases, the application must not use functionality that exceeds the version of Vulkan associated with a given object. The `pApiVersion` parameter returned by `vkEnumerateInstanceVersion` is the version associated with a `VkInstance` and its children, except for a `VkPhysicalDevice` and its children. `VkPhysicalDeviceProperties::apiVersion` is the version associated with a `VkPhysicalDevice` and its children.

**Note**
The encoding of **driverVersion** is implementation-defined. It may not use the same encoding as **apiVersion**. Applications should follow information from the vendor on how to extract the version information from **driverVersion**.
The `vendorID` and `deviceID` fields are provided to allow applications to adapt to device characteristics that are not adequately exposed by other Vulkan queries.

**Note**

These may include performance profiles, hardware errata, or other characteristics.

The `vendor` identified by `vendorID` is the entity responsible for the most salient characteristics of the underlying implementation of the `VkPhysicalDevice` being queried.

**Note**

For example, in the case of a discrete GPU implementation, this should be the GPU chipset vendor. In the case of a hardware accelerator integrated into a system-on-chip (SoC), this should be the supplier of the silicon IP used to create the accelerator.

If the vendor has a PCI vendor ID, the low 16 bits of `vendorID` must contain that PCI vendor ID, and the remaining bits must be set to zero. Otherwise, the value returned must be a valid Khronos vendor ID, obtained as described in the Vulkan Documentation and Extensions: Procedures and Conventions document in the section “Registering a Vendor ID with Khronos”. Khronos vendor IDs are allocated starting at 0x10000, to distinguish them from the PCI vendor ID namespace. Khronos vendor IDs are symbolically defined in the `VkVendorId` type.

The vendor is also responsible for the value returned in `deviceID`. If the implementation is driven primarily by a PCI device with a PCI device ID, the low 16 bits of `deviceID` must contain that PCI device ID, and the remaining bits must be set to zero. Otherwise, the choice of what values to return may be dictated by operating system or platform policies - but should uniquely identify both the device version and any major configuration options (for example, core count in the case of multicore devices).

**Note**

The same device ID should be used for all physical implementations of that device version and configuration. For example, all uses of a specific silicon IP GPU version and configuration should use the same device ID, even if those uses occur in different SoCs.

Khronos vendor IDs which may be returned in `VkPhysicalDeviceProperties::vendorID` are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkVendorId {
    VK_VENDOR_ID_VIV = 0x10001,
    VK_VENDOR_ID_VSI = 0x10002,
    VK_VENDOR_ID_KAZAN = 0x10003,
    VK_VENDOR_ID_CODEPLAY = 0x10004,
    VK_VENDOR_ID_MESA = 0x10005,
    VK_VENDOR_ID_POCL = 0x10006,
} VkVendorId;
```
Khronos vendor IDs may be allocated by vendors at any time. Only the latest canonical versions of this Specification, of the corresponding \texttt{vk.xml} API Registry, and of the corresponding \texttt{vulkan_core.h} header file \textbf{must} contain all reserved Khronos vendor IDs.

Only Khronos vendor IDs are given symbolic names at present. PCI vendor IDs returned by the implementation can be looked up in the PCI-SIG database.

The physical device types which \textbf{may} be returned in \texttt{VkPhysicalDeviceProperties::devicetype} are:

\begin{verbatim}
// Provided by VK_VERSION_1_0
typedef enum VkPhysicalDeviceType {
    VK_PHYSICAL_DEVICE_TYPE_OTHER = 0,
    VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU = 1,
    VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU = 2,
    VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU = 3,
    VK_PHYSICAL_DEVICE_TYPE_CPU = 4,
} VkPhysicalDeviceType;
\end{verbatim}

- \texttt{VK_PHYSICAL_DEVICE_TYPE_OTHER} - the device does not match any other available types.
- \texttt{VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU} - the device is typically one embedded in or tightly coupled with the host.
- \texttt{VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU} - the device is typically a separate processor connected to the host via an interlink.
- \texttt{VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU} - the device is typically a virtual node in a virtualization environment.
- \texttt{VK_PHYSICALDEVICE_TYPE_CPU} - the device is typically running on the same processors as the host.

The physical device type is advertised for informational purposes only, and does not directly affect the operation of the system. However, the device type \textbf{may} correlate with other advertised properties or capabilities of the system, such as how many memory heaps there are.

To query general properties of physical devices once enumerated, call:

\begin{verbatim}
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceProperties2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceProperties2* pProperties);
\end{verbatim}

or the equivalent command
void vkGetPhysicalDeviceProperties2KHR(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceProperties2* pProperties);

- **physicalDevice** is the handle to the physical device whose properties will be queried.
- **pProperties** is a pointer to a `VkPhysicalDeviceProperties2` structure in which properties are returned.

Each structure in `pProperties` and its `pNext` chain contains members corresponding to implementation-dependent properties, behaviors, or limits. `vkGetPhysicalDeviceProperties2` fills in each member to specify the corresponding value for the implementation.

### Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceProperties2-physicalDevice-parameter**
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- **VUID-vkGetPhysicalDeviceProperties2-pProperties-parameter**
  - `pProperties` must be a valid pointer to a `VkPhysicalDeviceProperties2` structure

The `VkPhysicalDeviceProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceProperties2 {
    VkStructureType sType;
    void* pNext;
    VkPhysicalDeviceProperties properties;
} VkPhysicalDeviceProperties2;
```

or the equivalent

```c
// Provided by VK_KHR_get_physical_device_properties2
typedef VkPhysicalDeviceProperties2 VkPhysicalDeviceProperties2KHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **properties** is a `VkPhysicalDeviceProperties` structure describing properties of the physical device. This structure is written with the same values as if it were written by `vkGetPhysicalDeviceProperties`.

The `pNext` chain of this structure is used to extend the structure with properties defined by extensions.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceProperties2-sType-sType**
  
  sType must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2**

- **VUID-VkPhysicalDeviceProperties2-pNext-pNext**
  
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of

  - VkPhysicalDeviceAccelerationStructurePropertiesKHR,
  - VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT,
  - VkPhysicalDeviceConservativeRasterizationPropertiesEXT,
  - VkPhysicalDeviceCooperativeMatrixPropertiesNV,
  - VkPhysicalDeviceCustomBorderColorPropertiesEXT,
  - VkPhysicalDeviceDepthStencilResolveProperties,
  - VkPhysicalDeviceDescriptorIndexingProperties,
  - VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV,
  - VkPhysicalDeviceDiscardRectanglePropertiesEXT, VkPhysicalDeviceDriverProperties,
  - VkPhysicalDeviceExternalMemoryHostPropertiesEXT,
  - VkPhysicalDeviceFloatControlsProperties,
  - VkPhysicalDeviceFragmentDensityMap2PropertiesEXT,
  - VkPhysicalDeviceFragmentDensityMapPropertiesEXT,
  - VkPhysicalDeviceFragmentShadingRateEnumsPropertiesNV,
  - VkPhysicalDeviceFragmentShadingRatePropertiesKHR, VkPhysicalDeviceIDProperties,
  - VkPhysicalDeviceInlineUniformBlockPropertiesEXT,
  - VkPhysicalDeviceLineRasterizationPropertiesEXT,
  - VkPhysicalDeviceMaintenance3Properties,
  - VkPhysicalDeviceMeshShaderPropertiesNV,
  - VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX,
  - VkPhysicalDeviceMultiviewProperties, VkPhysicalDevicePCIBusInfoPropertiesEXT,
  - VkPhysicalDevicePerformanceQueryPropertiesKHR,
  - VkPhysicalDevicePointClippingProperties,
  - VkPhysicalDevicePortabilitySubsetPropertiesKHR,
  - VkPhysicalDeviceProtectedMemoryProperties,
  - VkPhysicalDeviceProvokingVertexPropertiesEXT,
  - VkPhysicalDevicePushDescriptorPropertiesKHR,
  - VkPhysicalDeviceRayTracingPipelinePropertiesKHR,
  - VkPhysicalDeviceRayTracingPropertiesNV, VkPhysicalDeviceRobustness2PropertiesEXT,
  - VkPhysicalDeviceSampleLocationsPropertiesEXT,
  - VkPhysicalDeviceSamplerFilterMinmaxProperties,
  - VkPhysicalDeviceShaderCoreProperties2AMD,
  - VkPhysicalDeviceShaderCorePropertiesAMD,
  - VkPhysicalDeviceShaderSMBuiltinsPropertiesNV,
  - VkPhysicalDeviceShadingRateImagePropertiesNV, VkPhysicalDeviceSubgroupProperties,
  - VkPhysicalDeviceSubgroupSizeControlPropertiesEXT,
  - VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT,
  - VkPhysicalDeviceTimelineSemaphoreProperties,
  - VkPhysicalDeviceTransformFeedbackPropertiesEXT,
  - VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT.

or
The `sType` value of each struct in the `pNext` chain must be unique.

The `VkPhysicalDeviceIDProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceIDProperties {
    VkStructureType    sType;
    void*              pNext;
    uint8_t            deviceUUID[VK_UUID_SIZE];
    uint8_t            driverUUID[VK_UUID_SIZE];
    uint8_t            deviceLUID[VK_LUID_SIZE];
    uint32_t           deviceNodeMask;
    VkBool32           deviceLUIDValid;
} VkPhysicalDeviceIDProperties;
```

or the equivalent:

```c
// Provided by VK_KHR_external_memory_capabilities, VK_KHR_external_semaphore_capabilities, VK_KHR_external_fence_capabilities
typedef VkPhysicalDeviceIDProperties VkPhysicalDeviceIDPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `deviceUUID` is an array of `VK_UUID_SIZE` `uint8_t` values representing a universally unique identifier for the device.
- `driverUUID` is an array of `VK_UUID_SIZE` `uint8_t` values representing a universally unique identifier for the driver build in use by the device.
- `deviceLUID` is an array of `VK_LUID_SIZE` `uint8_t` values representing a locally unique identifier for the device.
- `deviceNodeMask` is a `uint32_t` bitfield identifying the node within a linked device adapter corresponding to the device.
- `deviceLUIDValid` is a boolean value that will be `VK_TRUE` if `deviceLUID` contains a valid LUID and `deviceNodeMask` contains a valid node mask, and `VK_FALSE` if they do not.

If the `VkPhysicalDeviceIDProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

`deviceUUID` must be immutable for a given device across instances, processes, driver APIs, driver versions, and system reboots.

Applications can compare the `driverUUID` value across instance and process boundaries, and can make similar queries in external APIs to determine whether they are capable of sharing memory.
objects and resources using them with the device.

deviceUUID and/or driverUUID must be used to determine whether a particular external object can be shared between driver components, where such a restriction exists as defined in the compatibility table for the particular object type:

- External memory handle types compatibility
- External semaphore handle types compatibility
- External fence handle types compatibility

If deviceLUIDValid is VK_FALSE, the values of deviceLUID and deviceNodeMask are undefined. If deviceLUIDValid is VK_TRUE and Vulkan is running on the Windows operating system, the contents of deviceLUID can be cast to an LUID object and must be equal to the locally unique identifier of a IDXGIAdapter1 object that corresponds to physicalDevice. If deviceLUIDValid is VK_TRUE, deviceNodeMask must contain exactly one bit. If Vulkan is running on an operating system that supports the Direct3D 12 API and physicalDevice corresponds to an individual device in a linked device adapter, deviceNodeMask identifies the Direct3D 12 node corresponding to physicalDevice. Otherwise, deviceNodeMask must be 1.

Note

Although they have identical descriptions, VkPhysicalDeviceIDProperties::deviceUUID may differ from VkPhysicalDeviceProperties2::pipelineCacheUUID. The former is intended to identify and correlate devices across API and driver boundaries, while the latter is used to identify a compatible device and driver combination to use when serializing and de-serializing pipeline state.

Implementations should return deviceUUID values which are likely to be unique even in the presence of multiple Vulkan implementations (such as a GPU driver and a software renderer; two drivers for different GPUs; or the same Vulkan driver running on two logically different devices).

Khronos' conformance testing can not guarantee that deviceUUID values are actually unique, so implementors should make their own best efforts to ensure this. In particular, hard-coded deviceUUID values, especially all-0 bits, should never be used.

A combination of values unique to the vendor, the driver, and the hardware environment can be used to provide a deviceUUID which is unique to a high degree of certainty. Some possible inputs to such a computation are:

- Information reported by vkGetPhysicalDeviceProperties
- PCI device ID (if defined)
- PCI bus ID, or similar system configuration information.
- Driver binary checksums.
While `VkPhysicalDeviceIDProperties::deviceUUID` is specified to remain consistent across driver versions and system reboots, it is not intended to be usable as a serializable persistent identifier for a device. It may change when a device is physically added to, removed from, or moved to a different connector in a system while that system is powered down. Further, there is no reasonable way to verify with conformance testing that a given device retains the same UUID in a given system across all driver versions supported in that system. While implementations should make every effort to report consistent device UUIDs across driver versions, applications should avoid relying on the persistence of this value for uses other than identifying compatible devices for external object sharing purposes.

### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES`

The `VkPhysicalDeviceDriverProperties` structure is defined as:

```c
typedef struct VkPhysicalDeviceDriverProperties {
    VkStructureType         sType;
    void*                   pNext;
    VkDriverId              driverID;
    char                    driverName[VK_MAX_DRIVER_NAME_SIZE];
    char                    driverInfo[VK_MAX_DRIVER_INFO_SIZE];
    VkConformanceVersion    conformanceVersion;
} VkPhysicalDeviceDriverProperties;
```

or the equivalent

```c
// Provided by VK_KHR_driver_properties
typedef VkPhysicalDeviceDriverProperties VkPhysicalDeviceDriverPropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **driverID** is a unique identifier for the driver of the physical device.
- **driverName** is an array of `VK_MAX_DRIVER_NAME_SIZE` char containing a null-terminated UTF-8 string which is the name of the driver.
- **driverInfo** is an array of `VK_MAX_DRIVER_INFO_SIZE` char containing a null-terminated UTF-8 string with additional information about the driver.
- **conformanceVersion** is the version of the Vulkan conformance test this driver is conformant against (see `VkConformanceVersion`).
If the `VkPhysicalDeviceDriverProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These are properties of the driver corresponding to a physical device.

driverID must be immutable for a given driver across instances, processes, driver versions, and system reboots.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceDriverProperties-sType-sType
  
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES`

Khronos driver IDs which may be returned in `VkPhysicalDeviceDriverProperties::driverID` are:
typedef enum VkDriverId {
    VK_DRIVER_ID_AMD_PROPRIETARY = 1,
    VK_DRIVER_ID_AMD_OPEN_SOURCE = 2,
    VK_DRIVER_ID_MESA_RADV = 3,
    VK_DRIVER_ID_NVIDIA_PROPRIETARY = 4,
    VK_DRIVER_ID_INTEL_PROPRIETARY_WINDOWS = 5,
    VK_DRIVER_ID_INTEL_OPEN_SOURCE_MESA = 6,
    VK_DRIVER_ID_IMAGINATION_PROPRIETARY = 7,
    VK_DRIVER_ID_QUALCOMM_PROPRIETARY = 8,
    VK_DRIVER_ID_ARM_PROPRIETARY = 9,
    VK_DRIVER_ID_GOOGLE_SWIFTSHADER = 10,
    VK_DRIVER_ID_GGP_PROPRIETARY = 11,
    VK_DRIVER_ID_BROADCOM_PROPRIETARY = 12,
    VK_DRIVER_ID_MESA_LLVMPIPE = 13,
    VK_DRIVER_ID_MOLTENVK = 14,
    VK_DRIVER_ID_COREAVI_PROPRIETARY = 15,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_AMD_PROPRIETARY_KHR = VK_DRIVER_ID_AMD_PROPRIETARY,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_AMD_OPEN_SOURCE_KHR = VK_DRIVER_ID_AMD_OPEN_SOURCE,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_MESA_RADV_KHR = VK_DRIVER_ID_MESA_RADV,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_NVIDIA_PROPRIETARY_KHR = VK_DRIVER_ID_NVIDIA_PROPRIETARY,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_INTEL_PROPRIETARY_WINDOWS_KHR = VK_DRIVER_ID_INTEL_PROPRIETARY_WINDOWS,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_INTEL_OPEN_SOURCE_MESA_KHR = VK_DRIVER_ID_INTEL_OPEN_SOURCE_MESA,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_IMAGINATION_PROPRIETARY_KHR = VK_DRIVER_ID_IMAGINATION_PROPRIETARY,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_QUALCOMM_PROPRIETARY_KHR = VK_DRIVER_ID_QUALCOMM_PROPRIETARY,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_ARM_PROPRIETARY_KHR = VK_DRIVER_ID_ARM_PROPRIETARY,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_GOOGLE_SWIFTSHADER_KHR = VK_DRIVER_ID_GOOGLE_SWIFTSHADER,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_GGP_PROPRIETARY_KHR = VK_DRIVER_ID_GGP_PROPRIETARY,
    // Provided by VK_KHR_driver_properties
    VK_DRIVER_ID_BROADCOM_PROPRIETARY_KHR = VK_DRIVER_ID_BROADCOM_PROPRIETARY,
} VkDriverId;

or the equivalent

// Provided by VK_KHR_driver_properties
typedef VkDriverId VkDriverIdKHR;
Khronos driver IDs may be allocated by vendors at any time. There may be multiple driver IDs for the same vendor, representing different drivers (for e.g. different platforms, proprietary or open source, etc.). Only the latest canonical versions of this Specification, of the corresponding *vk.xml* API Registry, and of the corresponding *vulkan_core.h* header file **must** contain all reserved Khronos driver IDs.

Only driver IDs registered with Khronos are given symbolic names. There **may be** unregistered driver IDs returned.

The conformance test suite version an implementation is compliant with is described with the *VkConformanceVersion* structure:

```c
typedef struct VkConformanceVersion {
  uint8_t    major;
  uint8_t    minor;
  uint8_t    subminor;
  uint8_t    patch;
} VkConformanceVersion;
```

or the equivalent

```c
// Provided by VK_KHR_driver_properties
typedef VkConformanceVersion VkConformanceVersionKHR;
```

- major is the major version number of the conformance test suite.
- minor is the minor version number of the conformance test suite.
- subminor is the subminor version number of the conformance test suite.
- patch is the patch version number of the conformance test suite.

The *VkPhysicalDevicePCIBusInfoPropertiesEXT* structure is defined as:

```c
// Provided by VK_EXT_pci_bus_info
typedef struct VkPhysicalDevicePCIBusInfoPropertiesEXT {
  VkStructureType    sType;
  void*              pNext;
  uint32_t           pciDomain;
  uint32_t           pciBus;
  uint32_t           pciDevice;
  uint32_t           pcifunction;
} VkPhysicalDevicePCIBusInfoPropertiesEXT;
```

- sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• pciDomain is the PCI bus domain.
• pciBus is the PCI bus identifier.
• pciDevice is the PCI device identifier.
• pciFunction is the PCI device function identifier.

If the VkPhysicalDevicePCIBusInfoPropertiesEXT structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

These are properties of the PCI bus information of a physical device.

Valid Usage (Implicit)

• VUID-VkPhysicalDevicePCIBusInfoPropertiesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PCI_BUS_INFO_PROPERTIES_EXT

To query properties of queues available on a physical device, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceQueueFamilyProperties(
  VkPhysicalDevice                      physicalDevice,
  uint32_t*                              pQueueFamilyPropertyCount,
  VkQueueFamilyProperties*               pQueueFamilyProperties);
```

• physicalDevice is the handle to the physical device whose properties will be queried.
• pQueueFamilyPropertyCount is a pointer to an integer related to the number of queue families available or queried, as described below.
• pQueueFamilyProperties is either NULL or a pointer to an array of VkQueueFamilyProperties structures.

If pQueueFamilyProperties is NULL, then the number of queue families available is returned in pQueueFamilyPropertyCount. Implementations must support at least one queue family. Otherwise, pQueueFamilyPropertyCount must point to a variable set by the user to the number of elements in the pQueueFamilyProperties array, and on return the variable is overwritten with the number of structures actually written to pQueueFamilyProperties. If pQueueFamilyPropertyCount is less than the number of queue families available, at most pQueueFamilyPropertyCount structures will be written.
Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceQueueFamilyProperties-physicalDevice-parameter**
  
  *physicalDevice* must be a valid *VkPhysicalDevice* handle

- **VUID-vkGetPhysicalDeviceQueueFamilyProperties-pQueueFamilyPropertyCount-parameter**
  
  *pQueueFamilyPropertyCount* must be a valid pointer to a *uint32_t* value

- **VUID-vkGetPhysicalDeviceQueueFamilyProperties-pQueueFamilyProperties-parameter**
  
  If the value referenced by *pQueueFamilyPropertyCount* is not 0, and *pQueueFamilyProperties* is not NULL, *pQueueFamilyProperties* must be a valid pointer to an array of *pQueueFamilyPropertyCount* *VkQueueFamilyProperties* structures

The *VkQueueFamilyProperties* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkQueueFamilyProperties {
    VkQueueFlags    queueFlags;
    uint32_t        queueCount;
    uint32_t        timestampValidBits;
    VkExtent3D      minImageTransferGranularity;
} VkQueueFamilyProperties;
```

- *queueFlags* is a bitmask of *VkQueueFlagBits* indicating capabilities of the queues in this queue family.

- *queueCount* is the unsigned integer count of queues in this queue family. Each queue family must support at least one queue.

- *timestampValidBits* is the unsigned integer count of meaningful bits in the timestamps written via *vkCmdWriteTimestamp2KHR* or *vkCmdWriteTimestamp*. The valid range for the count is 36..64 bits, or a value of 0, indicating no support for timestamps. Bits outside the valid range are guaranteed to be zeros.

- *minImageTransferGranularity* is the minimum granularity supported for image transfer operations on the queues in this queue family.

The value returned in *minImageTransferGranularity* has a unit of compressed texel blocks for images having a block-compressed format, and a unit of texels otherwise.

Possible values of *minImageTransferGranularity* are:

- (0,0,0) which indicates that only whole mip levels must be transferred using the image transfer operations on the corresponding queues. In this case, the following restrictions apply to all offset and extent parameters of image transfer operations:
  - The *x*, *y*, and *z* members of a *VkOffset3D* parameter must always be zero.
  - The *width*, *height*, and *depth* members of a *VkExtent3D* parameter must always match the width, height, and depth of the image subresource corresponding to the parameter, respectively.
• \((A_x, A_y, A_z)\) where \(A_x, A_y,\) and \(A_z\) are all integer powers of two. In this case the following restrictions apply to all image transfer operations:
  ◦ \(x, y, \) and \(z\) of a \texttt{VkOffset3D} parameter \textbf{must} be integer multiples of \(A_x, A_y,\) and \(A_z,\) respectively.
  ◦ \texttt{width} of a \texttt{VkExtent3D} parameter \textbf{must} be an integer multiple of \(A_x,\) or else \(x + \texttt{width}\) \textbf{must} equal the width of the image subresource corresponding to the parameter.
  ◦ \texttt{height} of a \texttt{VkExtent3D} parameter \textbf{must} be an integer multiple of \(A_y,\) or else \(y + \texttt{height}\) \textbf{must} equal the height of the image subresource corresponding to the parameter.
  ◦ \texttt{depth} of a \texttt{VkExtent3D} parameter \textbf{must} be an integer multiple of \(A_z,\) or else \(z + \texttt{depth}\) \textbf{must} equal the depth of the image subresource corresponding to the parameter.
  ◦ If the format of the image corresponding to the parameters is one of the block-compressed formats then for the purposes of the above calculations the granularity \textbf{must} be scaled up by the compressed texel block dimensions.

Queues supporting graphics and/or compute operations \textbf{must} report \((1,1,1)\) in \texttt{minImageTransferGranularity}, meaning that there are no additional restrictions on the granularity of image transfer operations for these queues. Other queues supporting image transfer operations are only \textbf{required} to support whole mip level transfers, thus \texttt{minImageTransferGranularity} for queues belonging to such queue families \textbf{may} be \((0,0,0)\).

The \textbf{Device Memory} section describes memory properties queried from the physical device.

For physical device feature queries see the \textbf{Features} chapter.

Bits which \textbf{may} be set in \texttt{VkQueueFamilyProperties::queueFlags} indicating capabilities of queues in a queue family are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,
    VK_QUEUE_COMPUTE_BIT = 0x00000002,
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
    // Provided by VK_VERSION_1_1
    VK_QUEUE_PROTECTED_BIT = 0x00000010,
    #ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_decode_queue
    VK_QUEUE_VIDEO_DECODE_BIT_KHR = 0x00000020,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_encode_queue
    VK_QUEUE_VIDEO_ENCODE_BIT_KHR = 0x00000040,
    #endif
} VkQueueFlagBits;
```

• \texttt{VK_QUEUE_GRAPHICS_BIT} \textbf{specifies} that queues in this queue family support graphics operations.
• **VK_QUEUE_COMPUTE_BIT** specifies that queues in this queue family support compute operations.

• **VK_QUEUE_TRANSFER_BIT** specifies that queues in this queue family support transfer operations.

• **VK_QUEUE_SPARSE_BINDING_BIT** specifies that queues in this queue family support sparse memory management operations (see Sparse Resources). If any of the sparse resource features are enabled, then at least one queue family must support this bit.

• **VK_QUEUE_VIDEO_DECODE_BIT_KHR** indicates that queues in this queue family support Video Decode operations.

• **VK_QUEUE_VIDEO_ENCODE_BIT_KHR** indicates that queues in this queue family support Video Encode operations.

• if **VK_QUEUE_PROTECTED_BIT** is set, then the queues in this queue family support the **VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT** bit. (see Protected Memory). If the physical device supports the protectedMemory feature, at least one of its queue families must support this bit.

If an implementation exposes any queue family that supports graphics operations, at least one queue family of at least one physical device exposed by the implementation must support both graphics and compute operations.

Furthermore, if the protected memory physical device feature is supported, then at least one queue family of at least one physical device exposed by the implementation must support graphics operations, compute operations, and protected memory operations.

> **Note**

All commands that are allowed on a queue that supports transfer operations are also allowed on a queue that supports either graphics or compute operations. Thus, if the capabilities of a queue family include **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**, then reporting the **VK_QUEUE_TRANSFER_BIT** capability separately for that queue family is optional.

For further details see Queues.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueueFlags;
```

**VkQueueFlags** is a bitmask type for setting a mask of zero or more **VkQueueFlagBits**.

To query properties of queues available on a physical device, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceQueueFamilyProperties2(  
    VkPhysicalDevice physicalDevice,  
    uint32_t* pQueueFamilyPropertyCount,  
    VkQueueFamilyProperties2* pQueueFamilyProperties);
```

or the equivalent command
void vkGetPhysicalDeviceQueueFamilyProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pQueueFamilyPropertyCount,
    VkQueueFamilyProperties2*                   pQueueFamilyProperties);

- physicalDevice is the handle to the physical device whose properties will be queried.
- pQueueFamilyPropertyCount is a pointer to an integer related to the number of queue families available or queried, as described in `vkGetPhysicalDeviceQueueFamilyProperties`.
- pQueueFamilyProperties is either NULL or a pointer to an array of `VkQueueFamilyProperties2` structures.

`vkGetPhysicalDeviceQueueFamilyProperties2` behaves similarly to `vkGetPhysicalDeviceQueueFamilyProperties`, with the ability to return extended information in a `pNext` chain of output structures.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceQueueFamilyProperties2-physicalDevice-parameter physicalDevice must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceQueueFamilyProperties2-pQueueFamilyPropertyCount-parameter pQueueFamilyPropertyCount must be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceQueueFamilyProperties2-pQueueFamilyProperties-parameter If the value referenced by `pQueueFamilyPropertyCount` is not 0, and `pQueueFamilyProperties` is not NULL, `pQueueFamilyProperties` must be a valid pointer to an array of `VkQueueFamilyProperties2` structures

The `VkQueueFamilyProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkQueueFamilyProperties2 {
    VkStructureType            sType;
    void*                      pNext;
    VkQueueFamilyProperties    queueFamilyProperties;
} VkQueueFamilyProperties2;
```

or the equivalent

```c
// Provided by VK_KHR_get_physical_device_properties2
typedef VkQueueFamilyProperties2 VkQueueFamilyProperties2KHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
• queueFamilyProperties is a VkQueueFamilyProperties structure which is populated with the same values as in vkGetPhysicalDeviceQueueFamilyProperties.

Valid Usage (Implicit)

• VUID-VkQueueFamilyProperties2-sType-sType
  sType must be VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2

• VUID-VkQueueFamilyProperties2-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkQueueFamilyCheckpointProperties2NV, VkQueueFamilyCheckpointPropertiesNV, or VkVideoQueueFamilyProperties2KHR

• VUID-VkQueueFamilyProperties2-sType-unique
  The sType value of each struct in the pNext chain must be unique

The VkQueueFamilyCheckpointProperties2NV structure is defined as:

```c
// Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
typedef struct VkQueueFamilyCheckpointProperties2NV {
  VkStructureType             sType;
  void*                       pNext;
  VkPipelineStageFlags2KHR    checkpointExecutionStageMask;
} VkQueueFamilyCheckpointProperties2NV;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• checkpointExecutionStageMask is a mask indicating which pipeline stages the implementation can execute checkpoint markers in.

Additional queue family information can be queried by setting VkQueueFamilyProperties2::pNext to point to a VkQueueFamilyCheckpointProperties2NV structure.

Valid Usage (Implicit)

• VUID-VkQueueFamilyCheckpointProperties2NV-sType-sType
  sType must be VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_2_NV

The VkQueueFamilyCheckpointPropertiesNV structure is defined as:

Chapter 5. Devices and Queues | 107
typedef struct VkQueueFamilyCheckpointPropertiesNV {
    VkStructureType sType;
    void* pNext;
    VkPipelineStageFlags checkpointExecutionStageMask;
} VkQueueFamilyCheckpointPropertiesNV;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **checkpointExecutionStageMask** is a mask indicating which pipeline stages the implementation can execute checkpoint markers in.

Additional queue family information can be queried by setting `VkQueueFamilyProperties2::pNext` to point to a `VkQueueFamilyCheckpointPropertiesNV` structure.

**Valid Usage (Implicit)**

- VUID-VkQueueFamilyCheckpointPropertiesNV-sType-sType
  
sType must be VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_NV

To enumerate the performance query counters available on a queue family of a physical device, call:

```c
// Provided by VK_KHR_performance_query
VkResult vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    uint32_t* pCounterCount,
    VkPerformanceCounterKHR* pCounters,
    VkPerformanceCounterDescriptionKHR* pCounterDescriptions);
```

- **physicalDevice** is the handle to the physical device whose queue family performance query counter properties will be queried.
- **queueFamilyIndex** is the index into the queue family of the physical device we want to get properties for.
- **pCounterCount** is a pointer to an integer related to the number of counters available or queried, as described below.
- **pCounters** is either NULL or a pointer to an array of `VkPerformanceCounterKHR` structures.
- **pCounterDescriptions** is either NULL or a pointer to an array of `VkPerformanceCounterDescriptionKHR` structures.

If **pCounters** is NULL and **pCounterDescriptions** is NULL, then the number of counters available is returned in **pCounterCount**. Otherwise, **pCounterCount** must point to a variable set by the user to the number of elements in the **pCounters**, **pCounterDescriptions**, or both arrays and on return the
variable is overwritten with the number of structures actually written out. If \( p\text{CounterCount} \) is less than the number of counters available, at most \( p\text{CounterCount} \) structures will be written, and \( \text{VK_INCOMPLETE} \) will be returned instead of \( \text{VK_SUCCESS} \), to indicate that not all the available counters were returned.

### Valid Usage (Implicit)

- VUID-vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR-physicalDevice-parameter
  - \( \text{physicalDevice} \) must be a valid \( \text{VkPhysicalDevice} \) handle
- VUID-vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR-pCounterCount-parameter
  - \( p\text{CounterCount} \) must be a valid pointer to a \( \text{uint32_t} \) value
- VUID-vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR-pCounters-parameter
  - If the value referenced by \( p\text{CounterCount} \) is not 0, and \( p\text{Counters} \) is not NULL, \( p\text{Counters} \) must be a valid pointer to an array of \( p\text{CounterCount} \) \( \text{VkPerformanceCounterKHR} \) structures
- VUID-vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR-pCounterDescriptions-parameter
  - If the value referenced by \( p\text{CounterCount} \) is not 0, and \( p\text{CounterDescriptions} \) is not NULL, \( p\text{CounterDescriptions} \) must be a valid pointer to an array of \( p\text{CounterCount} \) \( \text{VkPerformanceCounterDescriptionKHR} \) structures

### Return Codes

**Success**
- \( \text{VK_SUCCESS} \)
- \( \text{VK_INCOMPLETE} \)

**Failure**
- \( \text{VK_ERROR_OUT_OF_HOST_MEMORY} \)
- \( \text{VK_ERROR_OUT_OF_DEVICE_MEMORY} \)
- \( \text{VK_ERROR_INITIALIZATION_FAILED} \)

The \( \text{VkPerformanceCounterKHR} \) structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkPerformanceCounterKHR {
    VkStructureType                   sType;
    const void*                       pNext;
    VkPerformanceCounterUnitKHR       unit;
    VkPerformanceCounterScopeKHR      scope;
    VkPerformanceCounterStorageKHR    storage;
    uint8_t                           uuid[VK_UUID_SIZE];
} VkPerformanceCounterKHR;
```

- \( s\text{Type} \) is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `unit` is a `VkPerformanceCounterUnitKHR` specifying the unit that the counter data will record.
• `scope` is a `VkPerformanceCounterScopeKHR` specifying the scope that the counter belongs to.
• `storage` is a `VkPerformanceCounterStorageKHR` specifying the storage type that the counter’s data uses.
• `uuid` is an array of size `VK_UUID_SIZE`, containing 8-bit values that represent a universally unique identifier for the counter of the physical device.

### Valid Usage (Implicit)

- `VUID-VkPerformanceCounterKHR-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR`
- `VUID-VkPerformanceCounterKHR-pNext-pNext`
  - `pNext` must be `NULL`

Performance counters have an associated unit. This unit describes how to interpret the performance counter result.

The performance counter unit types which **may** be returned in `VkPerformanceCounterKHR::unit` are:

```c
typedef enum VkPerformanceCounterUnitKHR {
    VK_PERFORMANCE_COUNTER_UNIT_GENERIC_KHR = 0,
    VK_PERFORMANCE_COUNTER_UNIT_PERCENTAGE_KHR = 1,
    VK_PERFORMANCE_COUNTER_UNIT_NANOSECONDS_KHR = 2,
    VK_PERFORMANCE_COUNTER_UNIT_BYTES_KHR = 3,
    VK_PERFORMANCE_COUNTER_UNIT_BYTES_PER_SECOND_KHR = 4,
    VK_PERFORMANCE_COUNTER_UNIT_KELVIN_KHR = 5,
    VK_PERFORMANCE_COUNTER_UNIT_WATTS_KHR = 6,
    VK_PERFORMANCE_COUNTER_UNIT_VOLTS_KHR = 7,
    VK_PERFORMANCE_COUNTER_UNIT_AMPS_KHR = 8,
    VK_PERFORMANCE_COUNTER_UNIT_HERTZ_KHR = 9,
    VK_PERFORMANCE_COUNTER_UNIT_CYCLES_KHR = 10,
} VkPerformanceCounterUnitKHR;
```

- `VK_PERFORMANCE_COUNTER_UNIT_GENERIC_KHR` - the performance counter unit is a generic data point.
- `VK_PERFORMANCE_COUNTER_UNIT_PERCENTAGE_KHR` - the performance counter unit is a percentage (%).
- `VK_PERFORMANCE_COUNTER_UNIT_NANOSECONDS_KHR` - the performance counter unit is a value of nanoseconds (ns).
- `VK_PERFORMANCE_COUNTER_UNIT_BYTES_KHR` - the performance counter unit is a value of bytes.
- `VK_PERFORMANCE_COUNTER_UNIT_BYTES_PER_SECOND_KHR` - the performance counter unit is a value of bytes/s.
- **VK_PERFORMANCE_COUNTER_UNIT_KELVIN_KHR** - the performance counter unit is a temperature reported in Kelvin.
- **VK_PERFORMANCE_COUNTER_UNIT_WATTS_KHR** - the performance counter unit is a value of watts (W).
- **VK_PERFORMANCE_COUNTER_UNIT_VOLTS_KHR** - the performance counter unit is a value of volts (V).
- **VK_PERFORMANCE_COUNTER_UNIT_AMPS_KHR** - the performance counter unit is a value of amps (A).
- **VK_PERFORMANCE_COUNTER_UNIT_HERTZ_KHR** - the performance counter unit is a value of hertz (Hz).
- **VK_PERFORMANCE_COUNTER_UNIT_CYCLES_KHR** - the performance counter unit is a value of cycles.

Performance counters have an associated scope. This scope describes the granularity of a performance counter.

The performance counter scope types which **may** be returned in `VkPerformanceCounterKHR::scope` are:

```c
// Provided by VK_KHR_performance_query
typedef enum VkPerformanceCounterScopeKHR {
    VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR = 0,
    VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR = 1,
    VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_KHR = 2,
    VK_QUERY_SCOPE_COMMAND_BUFFER_KHR = VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR,
    VK_QUERY_SCOPE_RENDER_PASS_KHR = VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR,
    VK_QUERY_SCOPE_COMMAND_KHR = VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_KHR,
} VkPerformanceCounterScopeKHR;
```

- **VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR** - the performance counter scope is a single complete command buffer.
- **VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR** - the performance counter scope is zero or more complete render passes. The performance query containing the performance counter **must** begin and end outside a render pass instance.
- **VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_KHR** - the performance counter scope is zero or more commands.

Performance counters have an associated storage. This storage describes the payload of a counter result.

The performance counter storage types which **may** be returned in `VkPerformanceCounterKHR::storage` are:
// Provided by VK_KHR_performance_query
typedef enum VkPerformanceCounterStorageKHR {
    VK_PERFORMANCE_COUNTER_STORAGE_INT32_KHR = 0,
    VK_PERFORMANCE_COUNTER_STORAGE_INT64_KHR = 1,
    VK_PERFORMANCE_COUNTER_STORAGE_UINT32_KHR = 2,
    VK_PERFORMANCE_COUNTER_STORAGE_UINT64_KHR = 3,
    VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32_KHR = 4,
    VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64_KHR = 5,
} VkPerformanceCounterStorageKHR;

- **VK_PERFORMANCE_COUNTER_STORAGE_INT32_KHR** - the performance counter storage is a 32-bit signed integer.
- **VK_PERFORMANCE_COUNTER_STORAGE_INT64_KHR** - the performance counter storage is a 64-bit signed integer.
- **VK_PERFORMANCE_COUNTER_STORAGE_UINT32_KHR** - the performance counter storage is a 32-bit unsigned integer.
- **VK_PERFORMANCE_COUNTER_STORAGE_UINT64_KHR** - the performance counter storage is a 64-bit unsigned integer.
- **VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32_KHR** - the performance counter storage is a 32-bit floating-point.
- **VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64_KHR** - the performance counter storage is a 64-bit floating-point.

The VkPerformanceCounterDescriptionKHR structure is defined as:

// Provided by VK_KHR_performance_query
typedef struct VkPerformanceCounterDescriptionKHR {
    VkStructureType sType;
    const void* pNext;
    VkPerformanceCounterDescriptionFlagsKHR flags;
    char name[VK_MAX_DESCRIPTION_SIZE];
    char category[VK_MAX_DESCRIPTION_SIZE];
    char description[VK_MAX_DESCRIPTION_SIZE];
} VkPerformanceCounterDescriptionKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkPerformanceCounterDescriptionFlagBitsKHR indicating the usage behavior for the counter.
- **name** is an array of size VK_MAX_DESCRIPTION_SIZE, containing a null-terminated UTF-8 string specifying the name of the counter.
- **category** is an array of size VK_MAX_DESCRIPTION_SIZE, containing a null-terminated UTF-8 string specifying the category of the counter.
• description is an array of size VK_MAX_DESCRIPTION_SIZE, containing a null-terminated UTF-8 string specifying the description of the counter.

```
Valid Usage (Implicit)
```

- VUID-VkPerformanceCounterDescriptionKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR

- VUID-VkPerformanceCounterDescriptionKHR-pNext-pNext
  pNext must be NULL

Bits which can be set in VkPerformanceCounterDescriptionKHR::flags to specify usage behavior for a command pool are:

```c
// Provided by VK_KHR_performance_query
typedef enum VkPerformanceCounterDescriptionFlagBitsKHR {
    VK_PERFORMANCE_COUNTER_DESCRIPTION_PERFORMANCE_IMPACTING_BIT_KHR = 0x00000001,
    VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_BIT_KHR = 0x00000002,
} VkPerformanceCounterDescriptionFlagBitsKHR;
```

- VK_PERFORMANCE_COUNTER_DESCRIPTION_PERFORMANCE_IMPACTING_BIT_KHR specifies that recording the counter may have a noticeable performance impact.

- VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_BIT_KHR specifies that concurrently recording the counter while other submitted command buffers are running may impact the accuracy of the recording.

```c
// Provided by VK_KHR_performance_query
typedef VkFlags VkPerformanceCounterDescriptionFlagsKHR;
```

VkPerformanceCounterDescriptionFlagsKHR is a bitmask type for setting a mask of zero or more VkPerformanceCounterDescriptionFlagBitsKHR.

5.2. Devices

Device objects represent logical connections to physical devices. Each device exposes a number of queue families each having one or more queues. All queues in a queue family support the same operations.

As described in Physical Devices, a Vulkan application will first query for all physical devices in a system. Each physical device can then be queried for its capabilities, including its queue and queue family properties. Once an acceptable physical device is identified, an application will create a corresponding logical device. The created logical device is then the primary interface to the
physical device.

How to enumerate the physical devices in a system and query those physical devices for their queue family properties is described in the Physical Device Enumeration section above.

A single logical device can be created from multiple physical devices, if those physical devices belong to the same device group. A device group is a set of physical devices that support accessing each other’s memory and recording a single command buffer that can be executed on all the physical devices. Device groups are enumerated by calling vkEnumeratePhysicalDeviceGroups, and a logical device is created from a subset of the physical devices in a device group by passing the physical devices through VkDeviceGroupDeviceCreateInfo. For two physical devices to be in the same device group, they must support identical extensions, features, and properties.

Note
Physical devices in the same device group must be so similar because there are no rules for how different features/properties would interact. They must return the same values for nearly every invariant vkGetPhysicalDevice* feature, property, capability, etc., but could potentially differ for certain queries based on things like having a different display connected, or different compositor, etc.. The specification does not attempt to enumerate which state is in each category, because such a list would quickly become out of date.

To retrieve a list of the device groups present in the system, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkEnumeratePhysicalDeviceGroups(
    VkInstance instance,
    uint32_t* pPhysicalDeviceGroupCount,
    VkPhysicalDeviceGroupProperties* pPhysicalDeviceGroupProperties);
```

or the equivalent command

```c
// Provided by VK_KHR_device_group_creation
VkResult vkEnumeratePhysicalDeviceGroupsKHR(
    VkInstance instance,
    uint32_t* pPhysicalDeviceGroupCount,
    VkPhysicalDeviceGroupProperties* pPhysicalDeviceGroupProperties);
```

- instance is a handle to a Vulkan instance previously created with vkCreateInstance.
- pPhysicalDeviceGroupCount is a pointer to an integer related to the number of device groups available or queried, as described below.
- pPhysicalDeviceGroupProperties is either NULL or a pointer to an array of VkPhysicalDeviceGroupProperties structures.

If pPhysicalDeviceGroupProperties is NULL, then the number of device groups available is returned in pPhysicalDeviceGroupCount. Otherwise, pPhysicalDeviceGroupCount must point to a variable set by the
user to the number of elements in the `pPhysicalDeviceGroupProperties` array, and on return the variable is overwritten with the number of structures actually written to `pPhysicalDeviceGroupProperties`. If `pPhysicalDeviceGroupCount` is less than the number of device groups available, at most `pPhysicalDeviceGroupCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available device groups were returned.

Every physical device **must** be in exactly one device group.

### Valid Usage (Implicit)

- **instance** must be a valid `VkInstance` handle
- **pPhysicalDeviceGroupCount** must be a valid pointer to a `uint32_t` value
- If the value referenced by `pPhysicalDeviceGroupCount` is not 0, and `pPhysicalDeviceGroupProperties` is not NULL, `pPhysicalDeviceGroupProperties` must be a valid pointer to an array of `pPhysicalDeviceGroupCount` `VkPhysicalDeviceGroupProperties` structures

### Return Codes

#### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkPhysicalDeviceGroupProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceGroupProperties {
    VkStructureType     sType;
    void*               pNext;
    uint32_t            physicalDeviceCount;
    VkPhysicalDevice    physicalDevices[VK_MAX_DEVICE_GROUP_SIZE];
    VkBool32            subsetAllocation;
} VkPhysicalDeviceGroupProperties;
```
or the equivalent

```c
// Provided by VK_KHR_device_group_creation
typedef VkPhysicalDeviceGroupProperties VkPhysicalDeviceGroupPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `physicalDeviceCount` is the number of physical devices in the group.
- `physicalDevices` is an array of `VK_MAX_DEVICE_GROUP_SIZE` `VkPhysicalDevice` handles representing all physical devices in the group. The first `physicalDeviceCount` elements of the array will be valid.
- `subsetAllocation` specifies whether logical devices created from the group support allocating device memory on a subset of devices, via the `deviceMask` member of the `VkMemoryAllocateFlagsInfo`. If this is `VK_FALSE`, then all device memory allocations are made across all physical devices in the group. If `physicalDeviceCount` is 1, then `subsetAllocation` must be `VK_FALSE`.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceGroupProperties-sType-sType
  sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES`
- `VUID-VkPhysicalDeviceGroupProperties-pNext-pNext
  pNext` must be `NULL`

### 5.2.1. Device Creation

Logical devices are represented by `VkDevice` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkDevice)
```

A logical device is created as a connection to a physical device. To create a logical device, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateDevice(
    VkPhysicalDevice                            physicalDevice,
    const VkDeviceCreateInfo*                   pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDevice*                                   pDevice);
```

- `physicalDevice` must be one of the device handles returned from a call to `vkEnumeratePhysicalDevices` (see Physical Device Enumeration).
• **pCreateInfo** is a pointer to a `VkDeviceCreateInfo` structure containing information about how to create the device.

• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

• **pDevice** is a pointer to a handle in which the created `VkDevice` is returned.

`vkCreateDevice` verifies that extensions and features requested in the `ppEnabledExtensionNames` and `pEnabledFeatures` members of `pCreateInfo`, respectively, are supported by the implementation. If any requested extension is not supported, `vkCreateDevice` must return `VK_ERROR_EXTENSION_NOT_PRESENT`. If any requested feature is not supported, `vkCreateDevice` must return `VK_ERROR_FEATURE_NOT_PRESENT`. Support for extensions can be checked before creating a device by querying `vkEnumerateDeviceExtensionProperties`. Support for features can similarly be checked by querying `vkGetPhysicalDeviceFeatures`.

After verifying and enabling the extensions the `VkDevice` object is created and returned to the application.

Multiple logical devices can be created from the same physical device. Logical device creation may fail due to lack of device-specific resources (in addition to the other errors). If that occurs, `vkCreateDevice` will return `VK_ERROR_TOO_MANY_OBJECTS`.

---

### Valid Usage

- VUID-vkCreateDevice-ppEnabledExtensionNames-01387
  All required extensions for each extension in the `VkDeviceCreateInfo ::ppEnabledExtensionNames` list must also be present in that list

---

### Valid Usage (Implicit)

- VUID-vkCreateDevice-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkCreateDevice-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkDeviceCreateInfo` structure

- VUID-vkCreateDevice-pAllocator-parameter
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateDevice-pDevice-parameter
  `pDevice` must be a valid pointer to a `VkDevice` handle
Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_INITIALIZATION_FAILED
• VK_ERROR_EXTENSION_NOT_PRESENT
• VK_ERROR_FEATURE_NOT_PRESENT
• VK_ERROR_TOO_MANY_OBJECTS
• VK_ERROR_DEVICE_LOST

The VkDeviceCreateInfo structure is defined as:

```c
typedef struct VkDeviceCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkDeviceCreateFlags                flags;
    uint32_t                           queueCreateInfoCount;
    const VkDeviceQueueCreateInfo*     pQueueCreateInfos;
    uint32_t                           enabledLayerCount;
    const char* const*                 ppEnabledLayerNames;
    uint32_t                           enabledExtensionCount;
    const char* const*                 ppEnabledExtensionNames;
    const VkPhysicalDeviceFeatures*    pEnabledFeatures;
} VkDeviceCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **queueCreateInfoCount** is the unsigned integer size of the **pQueueCreateInfos** array. Refer to the Queue Creation section below for further details.
- **pQueueCreateInfos** is a pointer to an array of VkDeviceQueueCreateInfo structures describing the queues that are requested to be created along with the logical device. Refer to the Queue Creation section below for further details.
- **enabledLayerCount** is deprecated and ignored.
- **ppEnabledLayerNames** is deprecated and ignored. See Device Layer Deprecation.
- **enabledExtensionCount** is the number of device extensions to enable.
• **ppEnabledExtensionNames** is a pointer to an array of enabledExtensionCount null-terminated UTF-8 strings containing the names of extensions to enable for the created device. See the Extensions section for further details.

• **pEnabledFeatures** is NULL or a pointer to a VkPhysicalDeviceFeatures structure containing boolean indicators of all the features to be enabled. Refer to the Features section for further details.

**Valid Usage**

- **VUID-VkDeviceCreateInfo-queueFamilyIndex-02802**
  The queueFamilyIndex member of each element of pQueueCreateInfos must be unique within pQueueCreateInfos, except that two members can share the same queueFamilyIndex if one is a protected-capable queue and one is not a protected-capable queue.

- **VUID-VkDeviceCreateInfo-pNext-00373**
  If the pNext chain includes a VkPhysicalDeviceFeatures2 structure, then pEnabledFeatures must be NULL.

- **VUID-VkDeviceCreateInfo-ppEnabledExtensionNames-01840**
  ppEnabledExtensionNames must not contain VK_AMD_negative_viewport_height.

- **VUID-VkDeviceCreateInfo-ppEnabledExtensionNames-03328**
  ppEnabledExtensionNames must not contain both VK_KHR_buffer_device_address and VK_EXT_buffer_device_address.

- **VUID-VkDeviceCreateInfo-ppEnabledExtensionNames-04451**
  If the VK_KHR_portability_subset extension is included in pProperties of vkEnumerateDeviceExtensionProperties, ppEnabledExtensions must include "VK_KHR_portability_subset".

- **VUID-VkDeviceCreateInfo-shadingRateImage-04478**
  If shadingRateImage is enabled, pipelineFragmentShadingRate must not be enabled.

- **VUID-VkDeviceCreateInfo-shadingRateImage-04479**
  If shadingRateImage is enabled, primitiveFragmentShadingRate must not be enabled.

- **VUID-VkDeviceCreateInfo-shadingRateImage-04480**
  If shadingRateImage is enabled, attachmentFragmentShadingRate must not be enabled.

- **VUID-VkDeviceCreateInfo-fragmentDensityMap-04481**
  If fragmentDensityMap is enabled, pipelineFragmentShadingRate must not be enabled.

- **VUID-VkDeviceCreateInfo-fragmentDensityMap-04482**
  If fragmentDensityMap is enabled, primitiveFragmentShadingRate must not be enabled.

- **VUID-VkDeviceCreateInfo-fragmentDensityMap-04483**
  If fragmentDensityMap is enabled, attachmentFragmentShadingRate must not be enabled.
Valid Usage (Implicit)

- VUID-VkDeviceCreateInfo-sType-sType
  
  **sType must be** `VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO`

- VUID-VkDeviceCreateInfo-pNext-pNext
  

• VUID-VkDeviceCreateInfo-sType-unique

The sType value of each struct in the pNext chain must be unique, with the exception of structures of type VkDeviceDeviceMemoryReportCreateInfoEXT or VkDevicePrivateDataCreateInfoEXT

• VUID-VkDeviceCreateInfo-flags-zerobitmask

flags must be 0
• **VUID-VkDeviceCreateInfo-pQueueCreateInfos-parameter**
  
  `pQueueCreateInfos` **must** be a valid pointer to an array of `queueCreateInfoCount` valid `VkDeviceQueueCreateInfo` structures.

• **VUID-VkDeviceCreateInfo-ppEnabledLayerNames-parameter**
  
  If `enabledLayerCount` is not 0, `ppEnabledLayerNames` **must** be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings.

• **VUID-VkDeviceCreateInfo-ppEnabledExtensionNames-parameter**
  
  If `enabledExtensionCount` is not 0, `ppEnabledExtensionNames` **must** be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings.

• **VUID-VkDeviceCreateInfo-pEnabledFeatures-parameter**
  
  If `pEnabledFeatures` is not NULL, `pEnabledFeatures` **must** be a valid pointer to a valid `VkPhysicalDeviceFeatures` structure.

• **VUID-VkDeviceCreateInfo-queueCreateInfoCount-arraylength**
  
  `queueCreateInfoCount` **must** be greater than 0.

// Provided by VK_VERSION_1_0

typedef VkFlags VkDeviceCreateFlags;

**VkDeviceCreateFlags** is a bitmask type for setting a mask, but is currently reserved for future use.

A logical device **can** be created that connects to one or more physical devices by adding a `VkDeviceGroupDeviceCreateInfo` structure to the `pNext` chain of `VkDeviceCreateInfo`. The `VkDeviceGroupDeviceCreateInfo` structure is defined as:

// Provided by VK_VERSION_1_1

typedef struct VkDeviceGroupDeviceCreateInfo {
  VkStructureType sType;
  const void* pNext;
  uint32_t physicalDeviceCount;
  const VkPhysicalDevice* pPhysicalDevices;
} VkDeviceGroupDeviceCreateInfo;

or the equivalent

// Provided by VK_KHR_device_group_creation

typedef VkDeviceGroupDeviceCreateInfo VkDeviceGroupDeviceCreateInfoKHR;

• `sType` is the type of this structure.

• `pNext` is NULL or a pointer to a structure extending this structure.

• `physicalDeviceCount` is the number of elements in the `pPhysicalDevices` array.

• `pPhysicalDevices` is a pointer to an array of physical device handles belonging to the same device group.
The elements of the `pPhysicalDevices` array are an ordered list of the physical devices that the logical device represents. These must be a subset of a single device group, and need not be in the same order as they were enumerated. The order of the physical devices in the `pPhysicalDevices` array determines the `device index` of each physical device, with element i being assigned a device index of i. Certain commands and structures refer to one or more physical devices by using device indices or `device masks` formed using device indices.

A logical device created without using `VkDeviceGroupDeviceCreateInfo`, or with `physicalDeviceCount` equal to zero, is equivalent to a `physicalDeviceCount` of one and `pPhysicalDevices` pointing to the `physicalDevice` parameter to `vkCreateDevice`. In particular, the device index of that physical device is zero.

### Valid Usage

- **VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-00375**
  Each element of `pPhysicalDevices` must be unique

- **VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-00376**
  All elements of `pPhysicalDevices` must be in the same device group as enumerated by `vkEnumeratePhysicalDeviceGroups`

- **VUID-VkDeviceGroupDeviceCreateInfo-physicalDeviceCount-00377**
  If `physicalDeviceCount` is not 0, the `physicalDevice` parameter of `vkCreateDevice` must be an element of `pPhysicalDevices`

### Valid Usage (Implicit)

- **VUID-VkDeviceGroupDeviceCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO`

- **VUID-VkDeviceGroupDeviceCreateInfo-pPhysicalDevices-parameter**
  If `physicalDeviceCount` is not 0, `pPhysicalDevices` must be a valid pointer to an array of `physicalDeviceCount` valid `VkPhysicalDevice` handles

To specify whether device memory allocation is allowed beyond the size reported by `VkPhysicalDeviceMemoryProperties`, add a `VkDeviceMemoryOverallocationCreateInfoAMD` structure to the `pNext` chain of the `VkDeviceCreateInfo` structure. If this structure is not specified, it is as if the `VK_MEMORY_OVERALLOCATION_BEHAVIOR_DEFAULT_AMD` value is used.

```c
// Provided by VK_AMD_memory_overallocation_behavior
typedef struct VkDeviceMemoryOverallocationCreateInfoAMD {
    VkStructureType                      sType;
    const void*                          pNext;
    VkMemoryOverallocationBehaviorAMD    overallocationBehavior;
} VkDeviceMemoryOverallocationCreateInfoAMD;
```

- `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `overallocationBehavior` is the desired overallocation behavior.

Valid Usage (Implicit)

• `VUID-VkDeviceMemoryOverallocationCreateInfoAMD-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OVERALLOCATION_CREATE_INFO_AMD`

• `VUID-VkDeviceMemoryOverallocationCreateInfoAMD-overallocationBehavior-parameter`
  `overallocationBehavior` must be a valid `VkMemoryOverallocationBehaviorAMD` value

Possible values for `VkDeviceMemoryOverallocationCreateInfoAMD::overallocationBehavior` include:

```c
// Provided by VK_AMD_memory_overallocation_behavior
typedef enum VkMemoryOverallocationBehaviorAMD {
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_DEFAULT_AMD = 0,
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_ALLOWED_AMD = 1,
    VK_MEMORY_OVERALLOCATION_BEHAVIOR_DISALLOWED_AMD = 2,
} VkMemoryOverallocationBehaviorAMD;
```

• `VK_MEMORY_OVERALLOCATION_BEHAVIOR_DEFAULT_AMD` lets the implementation decide if overallocation is allowed.

• `VK_MEMORY_OVERALLOCATION_BEHAVIOR_ALLOWED_AMD` specifies overallocation is allowed if platform permits.

• `VK_MEMORY_OVERALLOCATION_BEHAVIOR_DISALLOWED_AMD` specifies the application is not allowed to allocate device memory beyond the heap sizes reported by `VkPhysicalDeviceMemoryProperties`. Allocations that are not explicitly made by the application within the scope of the Vulkan instance are not accounted for.

When using the Nsight™ Aftermath SDK, to configure how device crash dumps are created, add a `VkDeviceDiagnosticsConfigCreateInfoNV` structure to the `pNext` chain of the `VkDeviceCreateInfo` structure.

```c
// Provided by VK_NV_device_diagnostics_config
typedef struct VkDeviceDiagnosticsConfigCreateInfoNV {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDeviceDiagnosticsConfigFlagsNV    flags;
} VkDeviceDiagnosticsConfigCreateInfoNV;
```

• `sType` is the type of this structure.

• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `flags` is a bitmask of `VkDeviceDiagnosticsConfigFlagBitsNV` specifying additional parameters for configuring diagnostic tools.
Valid Usage (Implicit)

- **VUID-VkDeviceDiagnosticsConfigCreateInfoNV-sType-sType** 
  *sType must be* \texttt{VK_STRUCTURE_TYPE_DEVICE_DIAGNOSTICS_CONFIG_CREATE_INFO_NV} 

- **VUID-VkDeviceDiagnosticsConfigCreateInfoNV-flags-parameter** 
  *flags must be a valid combination of* \texttt{VkDeviceDiagnosticsConfigFlagBitsNV} values

Bits which 	extbf{can} be set in \texttt{VkDeviceDiagnosticsConfigCreateInfoNV::flags} include:

```c
typedef enum by VK_NV_device_diagnostics_config
VkDeviceDiagnosticsConfigFlagBitsNV {
    VK_DEVICE_DIAGNOSTICS_CONFIG_ENABLE_SHADER_DEBUG_INFO_BIT_NV = 0x00000001,
    VK_DEVICE_DIAGNOSTICS_CONFIG_ENABLE_RESOURCE_TRACKING_BIT_NV = 0x00000002,
    VKDEVICE_DIAGNOSTICS_CONFIG_ENABLE_AUTOMATIC_CHECKPOINTS_BIT_NV = 0x00000004,
} VkDeviceDiagnosticsConfigFlagBitsNV;
```

- **VK_DEVICE_DIAGNOSTICS_CONFIG_ENABLE_SHADER_DEBUG_INFO_BIT_NV** enables the generation of debug information for shaders.

- **VK_DEVICE_DIAGNOSTICS_CONFIG_ENABLE_RESOURCE_TRACKING_BIT_NV** enables driver side tracking of resources (images, buffers, etc.) used to augment the device fault information.

- **VK_DEVICE_DIAGNOSTICS_CONFIG_ENABLE_AUTOMATIC_CHECKPOINTS_BIT_NV** enables automatic insertion of diagnostic checkpoints for draw calls, dispatches, trace rays, and copies. The CPU call stack at the time of the command will be associated as the marker data for the automatically inserted checkpoints.

```c
typedef by VK_NV_device_diagnostics_config
VkFlags VkDeviceDiagnosticsConfigFlagsNV;
```

\texttt{VkDeviceDiagnosticsConfigFlagsNV} is a bitmask type for setting a mask of zero or more \texttt{VkDeviceDiagnosticsConfigFlagBitsNV}.

To register callbacks for underlying device memory events of type \texttt{VkDeviceMemoryReportEventTypeEXT}, add one or multiple \texttt{VkDeviceDeviceMemoryReportCreateInfoEXT} structures to the \texttt{pNext} chain of the \texttt{VkDeviceCreateInfo} structure.

```c
typedef struct by VK_EXT_device_memory_report
VkDeviceDeviceMemoryReportCreateInfoEXT {
    VkStructureType                        sType;
    const void*                            pNext;
    VkDeviceMemoryReportFlagsEXT           flags;
    PFN_vkDeviceMemoryReportCallbackEXT    pfnUserCallback;
    void*                                  pUserData;
} VkDeviceDeviceMemoryReportCreateInfoEXT;
```
• **sType** is the type of this structure.
• **pNext** is `NULL` or a pointer to a structure extending this structure.
• **flags** is 0 and reserved for future use.
• **PFN_vkDeviceMemoryReportCallbackEXT** is the application callback function to call.
• **pUserData** is user data to be passed to the callback.

The callback **may** be called from multiple threads simultaneously.

The callback **must** be called only once by the implementation when a `VkDeviceMemoryReportEventTypeEXT` event occurs.

**Note**
The callback could be called from a background thread other than the thread calling the Vulkan commands.

**Valid Usage (Implicit)**

- **VUID-VkDeviceDeviceMemoryReportCreateInfoEXT-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_DEVICE_MEMORY_REPORT_CREATE_INFO_EXT`
- **VUID-VkDeviceDeviceMemoryReportCreateInfoEXT-flags-zerobitmask**
  - `flags` **must** be `0`
- **VUID-VkDeviceDeviceMemoryReportCreateInfoEXT-pfnUserCallback-parameter**
  - `PFN_vkDeviceMemoryReportCallbackEXT` **must** be a valid `PFN_vkDeviceMemoryReportCallbackEXT` value
- **VUID-VkDeviceDeviceMemoryReportCreateInfoEXT-pUserData-parameter**
  - `pUserData` **must** be a pointer value

The prototype for the `VkDeviceDeviceMemoryReportCreateInfoEXT::PFN_vkDeviceMemoryReportCallbackEXT` function implemented by the application is:

```c
// Provided by VK_EXT_device_memory_report
typedef void (VKAPI_PTR *PFN_vkDeviceMemoryReportCallbackEXT)(
    const VkDeviceMemoryReportCallbackDataEXT* pCallbackData,
    void* pUserData);
```

- **pCallbackData** contains all the callback related data in the `VkDeviceMemoryReportCallbackDataEXT` structure.
- **pUserData** is the user data provided when the `VkDeviceDeviceMemoryReportCreateInfoEXT` was created.

The callback **must** not make calls to any Vulkan commands.

The definition of `VkDeviceMemoryReportCallbackDataEXT` is:
// Provided by VK_EXT_device_memory_report
typedef struct VkDeviceMemoryReportCallbackDataEXT {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemoryReportFlagsEXT flags;
    VkDeviceMemoryReportEventTypeEXT type;
    uint64_t memoryObjectId;
    VkDeviceSize size;
    VkObjectType objectType;
    uint64_t objectHandle;
    uint32_t heapIndex;
} VkDeviceMemoryReportCallbackDataEXT;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **flags** is 0 and reserved for future use.

• **type** is a VkDeviceMemoryReportEventTypeEXT type specifying the type of event reported in this VkDeviceMemoryReportCallbackDataEXT structure.

• **memoryObjectId** is the unique id for the underlying memory object as described below.

• **size** is the size of the memory object in bytes. If **type** is VK_DEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT, VK_DEVICE_MEMORY_REPORT_EVENT_TYPE_IMPORT_EXT or VK_DEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATION_FAILED_EXT, **size** is a valid VkDeviceSize value. Otherwise, **size** is undefined.

• **objectType** is a VkObjectType value specifying the type of the object associated with this device memory report event. If **type** is VK_DEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT, VKDEVICE_MEMORY_REPORT_EVENT_TYPE_FREE_EXT, VKDEVICE_MEMORY_REPORT_EVENT_TYPE_IMPORT_EXT, VKDEVICE_MEMORY_REPORT_EVENT_TYPE_UNIMPORT_EXT or VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATION_FAILED_EXT, **objectType** is a valid VkObjectType enum. Otherwise, **objectType** is undefined.

• **objectHandle** is the object this device memory report event is attributed to. If **type** is VK_DEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT, VKDEVICE_MEMORY_REPORT_EVENT_TYPE_FREE_EXT, VKDEVICE_MEMORY_REPORT_EVENT_TYPE_IMPORT_EXT or VKDEVICE_MEMORY_REPORT_EVENT_TYPE_UNIMPORT_EXT, **objectHandle** is a valid Vulkan handle of the type associated with **objectType** as defined in the VkObjectType and Vulkan Handle Relationship table. Otherwise, **objectHandle** is undefined.

• **heapIndex** describes which memory heap this device memory allocation is made from. If **type** is VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT or VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATION_FAILED_EXT, **heapIndex** corresponds to one of the valid heaps from the VkPhysicalDeviceMemoryProperties structure. Otherwise, **heapIndex** is undefined.

**memoryObjectId** is used to avoid double-counting on the same memory object.

If an internally-allocated device memory object or a VkDeviceMemory cannot be exported,
memoryObjectId must be unique in the VkDevice.

If an internally-allocated device memory object or a VkDeviceMemory supports being exported, memoryObjectId must be unique system wide.

If an internal device memory object or a VkDeviceMemory is backed by an imported external memory object, memoryObjectId must be unique system wide.

Implementor’s Note

If the heap backing an internally-allocated device memory cannot be used to back VkDeviceMemory, implementations can advertise that heap with no types.

Note

This structure should only be considered valid during the lifetime of the triggered callback.

For VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT and VKDEVICE_MEMORY_REPORT_EVENT_TYPE_IMPORT_EXT events, objectHandle usually will not yet exist when the application or tool receives the callback. objectHandle will only exist when the create or allocate call that triggered the event returns, and if the allocation or import ends up failing objectHandle won’t ever exist.

Valid Usage (Implicit)

- VUID-VkDeviceMemoryReportCallbackDataEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_MEMORY_REPORT_CALLBACK_DATA_EXT

- VUID-VkDeviceMemoryReportCallbackDataEXT-pNext-pNext
  pNext must be NULL

Possible values of VkDeviceMemoryReportCallbackDataEXT::type, specifying event types which cause the device driver to call the callback, are:

// Provided by VK_EXT_device_memory_report
typedef enum VK_EXT_device_memory_report
    VkDeviceMemoryReportEventTypeEXT {
        VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT = 0,
        VKDEVICE_MEMORY_REPORT_EVENT_TYPE_FREE_EXT = 1,
        VKDEVICE_MEMORY_REPORT_EVENT_TYPE_IMPORT_EXT = 2,
        VKDEVICE_MEMORY_REPORT_EVENT_TYPE_UNIMPORT_EXT = 3,
        VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATION_FAILED_EXT = 4,
} VkDeviceMemoryReportEventTypeEXT;

- VKDEVICE_MEMORY_REPORT_EVENT_TYPE_ALLOCATE_EXT specifies this event corresponds to the allocation of an internal device memory object or a VkDeviceMemory.

- VKDEVICE_MEMORY_REPORT_EVENT_TYPE_FREE_EXT specifies this event corresponds to the
deallocation of an internally-allocated device memory object or a \texttt{VkDeviceMemory}.

- \texttt{VK\_DEVICE\_MEMORY\_REPORT\_EVENT\_TYPE\_IMPORT\_EXT} specifies this event corresponds to the import of an external memory object.

- \texttt{VK\_DEVICE\_MEMORY\_REPORT\_EVENT\_TYPE\_UNIMPORT\_EXT} specifies this event is the release of an imported external memory object.

- \texttt{VK\_DEVICE\_MEMORY\_REPORT\_EVENT\_TYPE\_ALLOCATION\_FAILED\_EXT} specifies this event corresponds to the failed allocation of an internal device memory object or a \texttt{VkDeviceMemory}.

To reserve private data storage slots, add a \texttt{VkDevicePrivateDataCreateInfoEXT} structure to the \texttt{pNext} chain of the \texttt{VkDeviceCreateInfo} structure. Reserving slots in this manner is not strictly necessary, but doing so may improve performance.

```c
// Provided by VK\_EXT\_private\_data
typedef struct VkDevicePrivateDataCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t privateDataSlotRequestCount;
} VkDevicePrivateDataCreateInfoEXT;
```

- \texttt{sType} is the type of this structure.

- \texttt{pNext} is NULL or a pointer to a structure extending this structure.

- \texttt{privateDataSlotRequestCount} is the amount of slots to reserve.

**Valid Usage (Implicit)**

- \texttt{VUID-VkDevicePrivateDataCreateInfoEXT-sType-sType} \texttt{sType} must be \texttt{VK\_STRUCTURE\_TYPE\_DEVICE\_PRIVATE\_DATA\_CREATE\_INFO\_EXT}

### 5.2.2. Device Use

The following is a high-level list of \texttt{VkDevice} uses along with references on where to find more information:

- Creation of queues. See the Queues section below for further details.

- Creation and tracking of various synchronization constructs. See Synchronization and Cache Control for further details.

- Allocating, freeing, and managing memory. See Memory Allocation and Resource Creation for further details.

- Creation and destruction of command buffers and command buffer pools. See Command Buffers for further details.

- Creation, destruction, and management of graphics state. See Pipelines and Resource Descriptors, among others, for further details.
5.2.3. Lost Device

A logical device may become lost for a number of implementation-specific reasons, indicating that pending and future command execution may fail and cause resources and backing memory to become undefined.

*Note*

Typical reasons for device loss will include things like execution timing out (to prevent denial of service), power management events, platform resource management, implementation errors.

Applications not adhering to valid usage may also result in device loss being reported, however this is not guaranteed. Even if device loss is reported, the system may be in an unrecoverable state, and further usage of the API is still considered invalid.

When this happens, certain commands will return VK_ERROR_DEVICE_LOST. After any such event, the logical device is considered lost. It is not possible to reset the logical device to a non-lost state, however the lost state is specific to a logical device (VkDevice), and the corresponding physical device (VkPhysicalDevice) may be otherwise unaffected.

In some cases, the physical device may also be lost, and attempting to create a new logical device will fail, returning VK_ERROR_DEVICE_LOST. This is usually indicative of a problem with the underlying implementation, or its connection to the host. If the physical device has not been lost, and a new logical device is successfully created from that physical device, it must be in the non-lost state.

*Note*

Whilst logical device loss may be recoverable, in the case of physical device loss, it is unlikely that an application will be able to recover unless additional, unaffected physical devices exist on the system. The error is largely informational and intended only to inform the user that a platform issue has occurred, and should be investigated further. For example, underlying hardware may have developed a fault or become physically disconnected from the rest of the system. In many cases, physical device loss may cause other more serious issues such as the operating system crashing; in which case it may not be reported via the Vulkan API.

When a device is lost, its child objects are not implicitly destroyed and their handles are still valid. Those objects must still be destroyed before their parents or the device can be destroyed (see the Object Lifetime section). The host address space corresponding to device memory mapped using vkMapMemory is still valid, and host memory accesses to these mapped regions are still valid, but the contents are undefined. It is still legal to call any API command on the device and child objects.

Once a device is lost, command execution may fail, and commands that return a VkResult may return VK_ERROR_DEVICE_LOST. Commands that do not allow runtime errors must still operate correctly for valid usage and, if applicable, return valid data.

Commands that wait indefinitely for device execution (namely vkDeviceWaitIdle, vkQueueWaitIdle,
vkWaitForFences or vkAcquireNextImageKHR with a maximum `timeout`, and `vkGetQueryPoolResults` with the `VK_QUERY_RESULT_WAIT_BIT` bit set in `flags`) must return in finite time even in the case of a lost device, and return either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`. For any command that may return `VK_ERROR_DEVICE_LOST`, for the purpose of determining whether a command buffer is in the `pending state`, or whether resources are considered in-use by the device, a return value of `VK_ERROR_DEVICE_LOST` is equivalent to `VK_SUCCESS`.

The content of any external memory objects that have been exported from or imported to a lost device become undefined. Objects on other logical devices or in other APIs which are associated with the same underlying memory resource as the external memory objects on the lost device are unaffected other than their content becoming undefined. The layout of subresources of images on other logical devices that are bound to `VkDeviceMemory` objects associated with the same underlying memory resources as external memory objects on the lost device becomes `VK_IMAGE_LAYOUT_UNDEFINED`.

The state of `VkSemaphore` objects on other logical devices created by importing a semaphore payload with temporary permanence which was exported from the lost device is undefined. The state of `VkSemaphore` objects on other logical devices that permanently share a semaphore payload with a `VkSemaphore` object on the lost device is undefined, and remains undefined following any subsequent signal operations. Implementations must ensure pending and subsequently submitted wait operations on such semaphores behave as defined in Semaphore State Requirements For Wait Operations for external semaphores not in a valid state for a wait operation.

### 5.2.4. Device Destruction

To destroy a device, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyDevice(
    VkDevice device,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

To ensure that no work is active on the device, `vkDeviceWaitIdle` can be used to gate the destruction of the device. Prior to destroying a device, an application is responsible for destroying/freeing any Vulkan objects that were created using that device as the first parameter of the corresponding `vkCreate*` or `vkAllocate*` command.

**Note**

The lifetime of each of these objects is bound by the lifetime of the `VkDevice` object. Therefore, to avoid resource leaks, it is critical that an application explicitly free all of these resources prior to calling `vkDestroyDevice`.
Valid Usage

- VUID-vkDestroyDevice-device-00378
  All child objects created on device must have been destroyed prior to destroying device.

- VUID-vkDestroyDevice-device-00379
  If VkAllocationCallbacks were provided when device was created, a compatible set of callbacks must be provided here.

- VUID-vkDestroyDevice-device-00380
  If no VkAllocationCallbacks were provided when device was created, pAllocator must be NULL.

Valid Usage (Implicit)

- VUID-vkDestroyDevice-device-parameter
  If device is not NULL, device must be a valid VkDevice handle.

- VUID-vkDestroyDevice-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure.

Host Synchronization

- Host access to device must be externally synchronized.
- Host access to all VkQueue objects received from device must be externally synchronized.

5.3. Queues

5.3.1. Queue Family Properties

As discussed in the Physical Device Enumeration section above, the vkGetPhysicalDeviceQueueFamilyProperties command is used to retrieve details about the queue families and queues supported by a device.

Each index in the pQueueFamilyProperties array returned by vkGetPhysicalDeviceQueueFamilyProperties describes a unique queue family on that physical device. These indices are used when creating queues, and they correspond directly with the queueFamilyIndex that is passed to the vkCreateDevice command via the VkDeviceQueueCreateInfo structure as described in the Queue Creation section below.

Grouping of queue families within a physical device is implementation-dependent.
The general expectation is that a physical device groups all queues of matching capabilities into a single family. However, while implementations should do this, it is possible that a physical device may return two separate queue families with the same capabilities.

Once an application has identified a physical device with the queue(s) that it desires to use, it will create those queues in conjunction with a logical device. This is described in the following section.

5.3.2. Queue Creation

Creating a logical device also creates the queues associated with that device. The queues to create are described by a set of VkDeviceQueueCreateInfo structures that are passed to vkCreateDevice in pQueueCreateInfos.

Queues are represented by VkQueue handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkQueue)
```

The VkDeviceQueueCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDeviceQueueCreateInfo {  
    VkStructureType sType;  
    const void* pNext;  
    VkDeviceQueueCreateFlags flags;  
    uint32_t queueFamilyIndex;  
    uint32_t queueCount;  
    const float* pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask indicating behavior of the queue.
- queueFamilyIndex is an unsigned integer indicating the index of the queue family to create on this device. This index corresponds to the index of an element of the pQueueFamilyProperties array that was returned by vkGetPhysicalDeviceQueueFamilyProperties.
- queueCount is an unsigned integer specifying the number of queues to create in the queue family indicated by queueFamilyIndex.
- pQueuePriorities is a pointer to an array of queueCount normalized floating point values, specifying priorities of work that will be submitted to each created queue. See Queue Priority for more information.
Valid Usage

- VUID-VkDeviceQueueCreateInfo-queueFamilyIndex-00381
  queueFamilyIndex must be less than pQueueFamilyPropertyCount returned by 
  vkGetPhysicalDeviceQueueFamilyProperties

- VUID-VkDeviceQueueCreateInfo-queueCount-00382
  queueCount must be less than or equal to the queueCount member of the 
  VkQueueFamilyProperties structure, as returned by 
  vkGetPhysicalDeviceQueueFamilyProperties in the 
  pQueueFamilyProperties[queueFamilyIndex]

- VUID-VkDeviceQueueCreateInfo-pQueuePriorities-00383
  Each element of pQueuePriorities must be between 0.0 and 1.0 inclusive

- VUID-VkDeviceQueueCreateInfo-flags-02861
  If the protected memory feature is not enabled, the 
  VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT bit of flags must not be set

Valid Usage (Implicit)

- VUID-VkDeviceQueueCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO

- VUID-VkDeviceQueueCreateInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of 
  VkDeviceQueueGlobalPriorityCreateInfoEXT

- VUID-VkDeviceQueueCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkDeviceQueueCreateInfo-flags-parameter
  flags must be a valid combination of VkDeviceQueueCreateFlagBits values

- VUID-VkDeviceQueueCreateInfo-pQueuePriorities-parameter
  pQueuePriorities must be a valid pointer to an array of queueCount float values

- VUID-VkDeviceQueueCreateInfo-queueCount-arraylength
  queueCount must be greater than 0

Bits which can be set in VkDeviceQueueCreateInfo::flags to specify usage behavior of the queue are:

// Provided by VK_VERSION_1_0
typedef enum VkDeviceQueueCreateFlagBits {
    // Provided by VK_VERSION_1_1
    VK_DEVICE_QUEUE_CREATE_PROTECTED_BIT = 0x00000001,
} VkDeviceQueueCreateFlagBits;

- VK DEVICE QUEUE CREATE PROTECTED BIT specifies that the device queue is a protected-capable
queue.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkDeviceQueueCreateFlags;
```

`VkDeviceQueueCreateFlags` is a bitmask type for setting a mask of zero or more `VkDeviceQueueCreateFlagBits`.

A queue can be created with a system-wide priority by adding a `VkDeviceQueueGlobalPriorityCreateInfoEXT` structure to the `pNext` chain of `VkDeviceQueueCreateInfo`.

The `VkDeviceQueueGlobalPriorityCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_global_priority
typedef struct VkDeviceQueueGlobalPriorityCreateInfoEXT {
    VkStructureType             sType;
    const void*                 pNext;
    VkQueueGlobalPriorityEXT    globalPriority;
} VkDeviceQueueGlobalPriorityCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `globalPriority` is the system-wide priority associated to this queue as specified by `VkQueueGlobalPriorityEXT`.

A queue created without specifying `VkDeviceQueueGlobalPriorityCreateInfoEXT` will default to `VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT`.

**Valid Usage (Implicit)**

- `VUID-VkDeviceQueueGlobalPriorityCreateInfoEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT`

- `VUID-VkDeviceQueueGlobalPriorityCreateInfoEXT-globalPriority-parameter`
  - `globalPriority` must be a valid `VkQueueGlobalPriorityEXT` value

Possible values of `VkDeviceQueueGlobalPriorityCreateInfoEXT::globalPriority`, specifying a system-wide priority level are:
Priority values are sorted in ascending order. A comparison operation on the enum values can be used to determine the priority order.

- **VK_QUEUE_GLOBAL_PRIORITY_LOW_EXT** is below the system default. Useful for non-interactive tasks.
- **VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT** is the system default priority.
- **VK_QUEUE_GLOBAL_PRIORITY_HIGH_EXT** is above the system default.
- **VK_QUEUE_GLOBAL_PRIORITY_REALTIME_EXT** is the highest priority. Useful for critical tasks.

Queues with higher system priority may be allotted more processing time than queues with lower priority. An implementation may allow a higher-priority queue to starve a lower-priority queue until the higher-priority queue has no further commands to execute.

Priorities imply no ordering or scheduling constraints.

No specific guarantees are made about higher priority queues receiving more processing time or better quality of service than lower priority queues.

The global priority level of a queue takes precedence over the per-process queue priority (VkDeviceQueueCreateInfo::pQueuePriorities).

Abuse of this feature may result in starving the rest of the system of implementation resources. Therefore, the driver implementation may deny requests to acquire a priority above the default priority (VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT) if the caller does not have sufficient privileges. In this scenario VK_ERROR_NOT_PERMITTED_EXT is returned.

The driver implementation may fail the queue allocation request if resources required to complete the operation have been exhausted (either by the same process or a different process). In this scenario VK_ERROR_INITIALIZATION_FAILED is returned.

To retrieve a handle to a VkQueue object, call:

```c
// Provided by VK_VERSION_1_0
void vkGetDeviceQueue(
    VkDevice device,
    uint32_t queueFamilyIndex,
    uint32_t queueIndex,
    VkQueue* pQueue);
```

- **device** is the logical device that owns the queue.
• queueFamilyIndex is the index of the queue family to which the queue belongs.

• queueIndex is the index within this queue family of the queue to retrieve.

• pQueue is a pointer to a VkQueue object that will be filled with the handle for the requested queue.

vkGetDeviceQueue must only be used to get queues that were created with the flags parameter of VkDeviceQueueCreateInfo set to zero. To get queues that were created with a non-zero flags parameter use vkGetDeviceQueue2.

### Valid Usage

- VUID-vkGetDeviceQueue-queueFamilyIndex-00384
  queueFamilyIndex must be one of the queue family indices specified when device was created, via the VkDeviceQueueCreateInfo structure

- VUID-vkGetDeviceQueue-queueIndex-00385
  queueIndex must be less than the number of queues created for the specified queue family index when device was created, via the queueCount member of the VkDeviceQueueCreateInfo structure

- VUID-vkGetDeviceQueue-flags-01841
  VkDeviceQueueCreateInfo::flags must have been set to zero when device was created

### Valid Usage (Implicit)

- VUID-vkGetDeviceQueue-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetDeviceQueue-pQueue-parameter
  pQueue must be a valid pointer to a VkQueue handle

To retrieve a handle to a VkQueue object with specific VkDeviceQueueCreateFlags creation flags, call:

```c
// Provided by VK_VERSION_1_1
void vkGetDeviceQueue2(
    VkDevice                                    device,
    const VkDeviceQueueInfo2*                   pQueueInfo,
    VkQueue*                                    pQueue);
```

• device is the logical device that owns the queue.

• pQueueInfo is a pointer to a VkDeviceQueueInfo2 structure, describing the parameters used to create the device queue.

• pQueue is a pointer to a VkQueue object that will be filled with the handle for the requested queue.
Valid Usage (Implicit)

- VUID-vkGetDeviceQueue2-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkGetDeviceQueue2-pQueueInfo-parameter
  
  **pQueueInfo** must be a valid pointer to a valid **VkDeviceQueueInfo2** structure

- VUID-vkGetDeviceQueue2-pQueue-parameter
  
  **pQueue** must be a valid pointer to a **VkQueue** handle

The **VkDeviceQueueInfo2** structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceQueueInfo2 {
    VkStructureType             sType;
    const void*                 pNext;
    VkDeviceQueueCreateFlags    flags;
    uint32_t                    queueFamilyIndex;
    uint32_t                    queueIndex;
} VkDeviceQueueInfo2;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure. The **pNext** chain of **VkDeviceQueueInfo2** is used to provide additional image parameters to **vkGetDeviceQueue2**.
- **flags** is a **VkDeviceQueueCreateFlags** value indicating the flags used to create the device queue.
- **queueFamilyIndex** is the index of the queue family to which the queue belongs.
- **queueIndex** is the index within this queue family of the queue to retrieve.

The queue returned by **vkGetDeviceQueue2** must have the same **flags** value from this structure as that used at device creation time in a **VkDeviceQueueCreateInfo** instance. If no matching **flags** were specified at device creation time, then the handle returned in **pQueue** must be **NULL**.

Valid Usage

- VUID-VkDeviceQueueInfo2-queueFamilyIndex-01842
  
  **queueFamilyIndex** must be one of the queue family indices specified when **device** was created, via the **VkDeviceQueueCreateInfo** structure

- VUID-VkDeviceQueueInfo2-queueIndex-01843
  
  **queueIndex** must be less than the number of queues created for the specified queue family index and **VkDeviceQueueCreateFlags** member **flags** equal to this **flags** value when **device** was created, via the **queueCount** member of the **VkDeviceQueueCreateInfo** structure
5.3.3. Queue Family Index

The queue family index is used in multiple places in Vulkan in order to tie operations to a specific family of queues.

When retrieving a handle to the queue via `vkGetDeviceQueue`, the queue family index is used to select which queue family to retrieve the `VkQueue` handle from as described in the previous section.

When creating a `VkCommandPool` object (see Command Pools), a queue family index is specified in the `VkCommandPoolCreateInfo` structure. Command buffers from this pool can only be submitted on queues corresponding to this queue family.

When creating `VkImage` (see Images) and `VkBuffer` (see Buffers) resources, a set of queue families is included in the `VkImageCreateInfo` and `VkBufferCreateInfo` structures to specify the queue families that can access the resource.

When inserting a `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` (see Pipeline Barriers), a source and destination queue family index is specified to allow the ownership of a buffer or image to be transferred from one queue family to another. See the Resource Sharing section for details.

5.3.4. Queue Priority

Each queue is assigned a priority, as set in the `VkDeviceQueueCreateInfo` structures when creating the device. The priority of each queue is a normalized floating point value between 0.0 and 1.0, which is then translated to a discrete priority level by the implementation. Higher values indicate a higher priority, with 0.0 being the lowest priority and 1.0 being the highest.

Within the same device, queues with higher priority may be allotted more processing time than queues with lower priority. The implementation makes no guarantees with regards to ordering or scheduling among queues with the same priority, other than the constraints defined by any explicit synchronization primitives. The implementation makes no guarantees with regards to queues across different devices.

An implementation may allow a higher-priority queue to starve a lower-priority queue on the same `VkDevice` until the higher-priority queue has no further commands to execute. The relationship of queue priorities must not cause queues on one `VkDevice` to starve queues on another `VkDevice`.

No specific guarantees are made about higher priority queues receiving more processing time or
better quality of service than lower priority queues.

5.3.5. Queue Submission

Work is submitted to a queue via *queue submission* commands such as *vkQueueSubmit2KHR* or *vkQueueSubmit*. Queue submission commands define a set of *queue operations* to be executed by the underlying physical device, including synchronization with semaphores and fences.

Submission commands take as parameters a target queue, zero or more *batches* of work, and an *optional* fence to signal upon completion. Each batch consists of three distinct parts:

1. Zero or more semaphores to wait on before execution of the rest of the batch.
   - If present, these describe a *semaphore wait operation*.
2. Zero or more work items to execute.
   - If present, these describe a *queue operation* matching the work described.
3. Zero or more semaphores to signal upon completion of the work items.
   - If present, these describe a *semaphore signal operation*.

If a fence is present in a queue submission, it describes a *fence signal operation*.

All work described by a queue submission command *must* be submitted to the queue before the command returns.

**Sparse Memory Binding**

In Vulkan it is possible to sparsely bind memory to buffers and images as described in the *Sparse Resource* chapter. Sparse memory binding is a queue operation. A queue whose flags include the *VK_QUEUE_SPARSE_BINDING_BIT* *must* be able to support the mapping of a virtual address to a physical address on the device. This causes an update to the page table mappings on the device. This update *must* be synchronized on a queue to avoid corrupting page table mappings during execution of graphics commands. By binding the sparse memory resources on queues, all commands that are dependent on the updated bindings are synchronized to only execute after the binding is updated. See the *Synchronization and Cache Control* chapter for how this synchronization is accomplished.

5.3.6. Queue Destruction

Queues are created along with a logical device during *vkCreateDevice*. All queues associated with a logical device are destroyed when *vkDestroyDevice* is called on that device.
Chapter 6. Command Buffers

Command buffers are objects used to record commands which can be subsequently submitted to a device queue for execution. There are two levels of command buffers - primary command buffers, which can execute secondary command buffers, and which are submitted to queues, and secondary command buffers, which can be executed by primary command buffers, and which are not directly submitted to queues.

Command buffers are represented by `VkCommandBuffer` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_HANDLE(VkCommandBuffer)
```

Recorded commands include commands to bind pipelines and descriptor sets to the command buffer, commands to modify dynamic state, commands to draw (for graphics rendering), commands to dispatch (for compute), commands to execute secondary command buffers (for primary command buffers only), commands to copy buffers and images, and other commands.

Each command buffer manages state independently of other command buffers. There is no inheritance of state across primary and secondary command buffers, or between secondary command buffers. When a command buffer begins recording, all state in that command buffer is undefined. When secondary command buffer(s) are recorded to execute on a primary command buffer, the secondary command buffer inherits no state from the primary command buffer, and all state of the primary command buffer is undefined after an execute secondary command buffer command is recorded. There is one exception to this rule - if the primary command buffer is inside a render pass instance, then the render pass and subpass state is not disturbed by executing secondary command buffers. For state dependent commands (such as draws and dispatches), any state consumed by those commands must not be undefined.

`VkCommandBufferInheritanceViewportScissorInfoNV` defines an exception allowing limited inheritance of dynamic viewport and scissor state.

Unless otherwise specified, and without explicit synchronization, the various commands submitted to a queue via command buffers may execute in arbitrary order relative to each other, and/or concurrently. Also, the memory side effects of those commands may not be directly visible to other commands without explicit memory dependencies. This is true within a command buffer, and across command buffers submitted to a given queue. See the synchronization chapter for information on implicit and explicit synchronization between commands.

6.1. Command Buffer Lifecycle

Each command buffer is always in one of the following states:

**Initial**

When a command buffer is allocated, it is in the initial state. Some commands are able to reset a command buffer (or a set of command buffers) back to this state from any of the executable, recording or invalid state. Command buffers in the initial state can only be moved to the
recording state, or freed.

**Recording**

vkBeginCommandBuffer changes the state of a command buffer from the initial state to the *recording state*. Once a command buffer is in the recording state, vkCmd* commands can be used to record to the command buffer.

**Executable**

vkEndCommandBuffer ends the recording of a command buffer, and moves it from the recording state to the *executable state*. Executable command buffers can be submitted, reset, or recorded to another command buffer.

**Pending**

Queue submission of a command buffer changes the state of a command buffer from the executable state to the *pending state*. Whilst in the pending state, applications must not attempt to modify the command buffer in any way - as the device may be processing the commands recorded to it. Once execution of a command buffer completes, the command buffer either reverts back to the executable state, or if it was recorded with **VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT**, it moves to the invalid state. A synchronization command should be used to detect when this occurs.

**Invalid**

Some operations, such as modifying or deleting a resource that was used in a command recorded to a command buffer, will transition the state of that command buffer into the *invalid state*. Command buffers in the invalid state can only be reset or freed.

![Figure 1. Lifecycle of a command buffer](image)

Any given command that operates on a command buffer has its own requirements on what state a command buffer must be in, which are detailed in the valid usage constraints for that command.

Resetting a command buffer is an operation that discards any previously recorded commands and puts a command buffer in the initial state. Resetting occurs as a result of vkResetCommandBuffer or vkResetCommandPool, or as part of vkBeginCommandBuffer (which additionally puts the command buffer in the recording state).
Secondary command buffers can be recorded to a primary command buffer via `vkCmdExecuteCommands`. This partially ties the lifecycle of the two command buffers together - if the primary is submitted to a queue, both the primary and any secondaries recorded to it move to the pending state. Once execution of the primary completes, so it does for any secondary recorded within it. After all executions of each command buffer complete, they each move to their appropriate completion state (either to the execution state or the invalid state, as specified above).

If a secondary moves to the invalid state or the initial state, then all primary buffers it is recorded in move to the invalid state. A primary moving to any other state does not affect the state of a secondary recorded in it.

```
Note
Resetting or freeing a primary command buffer removes the lifecycle linkage to all secondary command buffers that were recorded into it.
```

### 6.2. Command Pools

Command pools are opaque objects that command buffer memory is allocated from, and which allow the implementation to amortize the cost of resource creation across multiple command buffers. Command pools are externally synchronized, meaning that a command pool must not be used concurrently in multiple threads. That includes use via recording commands on any command buffers allocated from the pool, as well as operations that allocate, free, and reset command buffers or the pool itself.

Command pools are represented by `VkCommandPool` handles:

```
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkCommandPool)
```

To create a command pool, call:

```
// Provided by VK_VERSION_1_0
VkResult vkCreateCommandPool(  
VkDevice device,  
const VkCommandPoolCreateInfo* pCreateInfo,  
const VkAllocationCallbacks* pAllocator,  
VkCommandPool* pCommandPool);
```

- `device` is the logical device that creates the command pool.
- `pCreateInfo` is a pointer to a `VkCommandPoolCreateInfo` structure specifying the state of the command pool object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pCommandPool` is a pointer to a `VkCommandPool` handle in which the created pool is returned.
Valid Usage

- VUID-vkCreateCommandPool-queueFamilyIndex-01937
  `pCreateInfo->queueFamilyIndex` **must** be the index of a queue family available in the logical device `device`

Valid Usage (Implicit)

- VUID-vkCreateCommandPool-device-parameter
  `device` **must** be a valid `VkDevice` handle

- VUID-vkCreateCommandPool-pCreateInfo-parameter
  `pCreateInfo` **must** be a valid pointer to a valid `VkCommandPoolCreateInfo` structure

- VUID-vkCreateCommandPool-pAllocator-parameter
  If `pAllocator` is not NULL, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateCommandPool-pCommandPool-parameter
  `pCommandPool` **must** be a valid pointer to a `VkCommandPool` handle

Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkCommandPoolCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCommandPoolCreateInfo {
    VkStructureType             sType;  // Type of this structure.
    const void*                 pNext;  // NULL or a pointer to a structure extending this structure.
    VkCommandPoolCreateFlags    flags;  // A bitmask of `VkCommandPoolCreateFlagBits` indicating usage behavior for the pool and command buffers allocated from it.
    uint32_t                     queueFamilyIndex;  // Queue family index.
} VkCommandPoolCreateInfo;
```
• queueFamilyIndex designates a queue family as described in section Queue Family Properties. All command buffers allocated from this command pool must be submitted on queues from the same queue family.

Valid Usage

• VUID-VkCommandPoolCreateInfo-flags-02860
  If the protected memory feature is not enabled, the VK_COMMAND_POOL_CREATE_PROTECTED_BIT bit of flags must not be set.

Valid Usage (Implicit)

• VUID-VkCommandPoolCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO.

• VUID-VkCommandPoolCreateInfo-pNext-pNext
  pNext must be NULL.

• VUID-VkCommandPoolCreateInfo-flags-parameter
  flags must be a valid combination of VkCommandPoolCreateFlagBits values.

Bits which can be set in VkCommandPoolCreateInfo::flags to specify usage behavior for a command pool are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandPoolCreateFlagBits {
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001,
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002,
// Provided by VK_VERSION_1_1
    VK_COMMAND_POOL_CREATE_PROTECTED_BIT = 0x00000004,
} VkCommandPoolCreateFlagBits;
```

• VK_COMMAND_POOL_CREATE_TRANSIENT_BIT specifies that command buffers allocated from the pool will be short-lived, meaning that they will be reset or freed in a relatively short timeframe. This flag may be used by the implementation to control memory allocation behavior within the pool.

• VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT allows any command buffer allocated from a pool to be individually reset to the initial state; either by calling vkResetCommandBuffer, or via the implicit reset when calling vkBeginCommandBuffer. If this flag is not set on a pool, then vkResetCommandBuffer must not be called for any command buffer allocated from that pool.

• VK_COMMAND_POOL_CREATE_PROTECTED_BIT specifies that command buffers allocated from the pool are protected command buffers.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandPoolCreateFlags;
```
VkCommandPoolCreateFlags is a bitmask type for setting a mask of zero or more VkCommandPoolCreateFlagBits.

To trim a command pool, call:

```c
// Provided by VK_VERSION_1_1
void vkTrimCommandPool(
    VkDevice device,
    VkCommandPool commandPool,
    VkCommandPoolTrimFlags flags);
```
or the equivalent command

```c
// Provided by VK_KHR_maintenance1
void vkTrimCommandPoolKHR(
    VkDevice device,
    VkCommandPool commandPool,
    VkCommandPoolTrimFlags flags);
```

- **device** is the logical device that owns the command pool.
- **commandPool** is the command pool to trim.
- **flags** is reserved for future use.

Trimming a command pool recycles unused memory from the command pool back to the system. Command buffers allocated from the pool are not affected by the command.
Note

This command provides applications with some control over the internal memory allocations used by command pools.

Unused memory normally arises from command buffers that have been recorded and later reset, such that they are no longer using the memory. On reset, a command buffer can return memory to its command pool, but the only way to release memory from a command pool to the system requires calling `vkResetCommandPool`, which cannot be executed while any command buffers from that pool are still in use. Subsequent recording operations into command buffers will re-use this memory but since total memory requirements fluctuate over time, unused memory can accumulate.

In this situation, trimming a command pool may be useful to return unused memory back to the system, returning the total outstanding memory allocated by the pool back to a more “average” value.

Implementations utilize many internal allocation strategies that make it impossible to guarantee that all unused memory is released back to the system. For instance, an implementation of a command pool may involve allocating memory in bulk from the system and sub-allocating from that memory. In such an implementation any live command buffer that holds a reference to a bulk allocation would prevent that allocation from being freed, even if only a small proportion of the bulk allocation is in use.

In most cases trimming will result in a reduction in allocated but unused memory, but it does not guarantee the “ideal” behavior.

Trimming may be an expensive operation, and should not be called frequently. Trimming should be treated as a way to relieve memory pressure after application-known points when there exists enough unused memory that the cost of trimming is “worth” it.

Valid Usage (Implicit)

- `VK_KHR_pipeline_executed_directly` must be a valid `VkDevice` handle
- `commandPool` must be a valid `VkCommandPool` handle
- `flags` must be 0
- `commandPool` must have been created, allocated, or retrieved from `device`
Host Synchronization

- Host access to `commandPool` must be externally synchronized

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkCommandPoolTrimFlags;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance1
typedef VkCommandPoolTrimFlags VkCommandPoolTrimFlagsKHR;
```

`VkCommandPoolTrimFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To reset a command pool, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetCommandPool(
    VkDevice                                    device,
    VkCommandPool                               commandPool,
    VkCommandPoolResetFlags                     flags);
```

- `device` is the logical device that owns the command pool.
- `commandPool` is the command pool to reset.
- `flags` is a bitmask of `VkCommandPoolResetFlagBits` controlling the reset operation.

Resetting a command pool recycles all of the resources from all of the command buffers allocated from the command pool back to the command pool. All command buffers that have been allocated from the command pool are put in the initial state.

Any primary command buffer allocated from another `VkCommandPool` that is in the recording or executable state and has a secondary command buffer allocated from `commandPool` recorded into it, becomes invalid.

Valid Usage

- `VUID-vkResetCommandPool-commandPool-00040`
  All `VkCommandBuffer` objects allocated from `commandPool` must not be in the pending state
Valid Usage (Implicit)

- VUID-vkResetCommandPool-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkResetCommandPool-commandPool-parameter
  
  `commandPool` must be a valid `VkCommandPool` handle

- VUID-vkResetCommandPool-flags-parameter
  
  `flags` must be a valid combination of `VkCommandPoolResetFlagBits` values

- VUID-vkResetCommandPool-commandPool-parent
  
  `commandPool` must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `commandPool` must be externally synchronized

Return Codes

Success

- `VK_SUCCESS`

Failure

- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Bits which can be set in `vkResetCommandPool::flags` to control the reset operation are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkCommandPoolResetFlagBits {
    VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
} VkCommandPoolResetFlagBits;
```

- `VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT` specifies that resetting a command pool recycles all of the resources from the command pool back to the system.

```cpp
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandPoolResetFlags;
```

`VkCommandPoolResetFlags` is a bitmask type for setting a mask of zero or more `VkCommandPoolResetFlagBits`.

To destroy a command pool, call:
void vkDestroyCommandPool(
  VkDevice device,  
  VkCommandPool commandPool,  
  const VkAllocationCallbacks* pAllocator);

- **device** is the logical device that destroys the command pool.
- **commandPool** is the handle of the command pool to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

When a pool is destroyed, all command buffers allocated from the pool are **freed**.

Any primary command buffer allocated from another **VkCommandPool** that is in the recording or executable state and has a secondary command buffer allocated from **commandPool** recorded into it, becomes **invalid**.

### Valid Usage

- **VUID-vkDestroyCommandPool-commandPool-00041**
  All **VkCommandBuffer** objects allocated from **commandPool** must not be in the pending state

- **VUID-vkDestroyCommandPool-commandPool-00042**
  If **VkAllocationCallbacks** were provided when **commandPool** was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyCommandPool-commandPool-00043**
  If no **VkAllocationCallbacks** were provided when **commandPool** was created, **pAllocator** must be **NULL**

### Valid Usage (Implicit)

- **VUID-vkDestroyCommandPool-device-parameter**
  **device** must be a valid **VkDevice** handle

- **VUID-vkDestroyCommandPool-commandPool-parameter**
  If **commandPool** is not **VK_NULL_HANDLE**, **commandPool** must be a valid **VkCommandPool** handle

- **VUID-vkDestroyCommandPool-pAllocator-parameter**
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- **VUID-vkDestroyCommandPool-commandPool-parent**
  If **commandPool** is a valid handle, it must have been created, allocated, or retrieved from **device**
6.3. Command Buffer Allocation and Management

To allocate command buffers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateCommandBuffers(
    VkDevice                                    device,
    const VkCommandBufferAllocateInfo*          pAllocateInfo,
    VkCommandBuffer*                            pCommandBuffers);
```

- `device` is the logical device that owns the command pool.
- `pAllocateInfo` is a pointer to a `VkCommandBufferAllocateInfo` structure describing parameters of the allocation.
- `pCommandBuffers` is a pointer to an array of `VkCommandBuffer` handles in which the resulting command buffer objects are returned. The array must be at least the length specified by the `commandBufferCount` member of `pAllocateInfo`. Each allocated command buffer begins in the initial state.

`vkAllocateCommandBuffers` can be used to create multiple command buffers. If the creation of any of those command buffers fails, the implementation must destroy all successfully created command buffer objects from this command, set all entries of the `pCommandBuffers` array to `NULL` and return the error.

When command buffers are first allocated, they are in the initial state.

Valid Usage (Implicit)

- VUID-vkAllocateCommandBuffers-device-parameter
  device must be a valid `VkDevice` handle

- VUID-vkAllocateCommandBuffers-pAllocateInfo-parameter
  `pAllocateInfo` must be a valid pointer to a valid `VkCommandBufferAllocateInfo` structure

- VUID-vkAllocateCommandBuffers-pCommandBuffers-parameter
  `pCommandBuffers` must be a valid pointer to an array of `pAllocateInfo->commandBufferCount` `VkCommandBuffer` handles

- VUID-vkAllocateCommandBuffers-pAllocateInfo::commandBufferCount-arraylength
  `pAllocateInfo->commandBufferCount` must be greater than 0
Host Synchronization

- Host access to `pAllocateInfo->commandPool` must be externally synchronized.

Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The `VkCommandBufferAllocateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCommandBufferAllocateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkCommandPool           commandPool;
    VkCommandBufferLevel    level;
    uint32_t                commandBufferCount;
} VkCommandBufferAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `commandPool` is the command pool from which the command buffers are allocated.
- `level` is a `VkCommandBufferLevel` value specifying the command buffer level.
- `commandBufferCount` is the number of command buffers to allocate from the pool.

Valid Usage (Implicit)

- **VUID-VkCommandBufferAllocateInfo-sType-sType**
  `sType` must be **VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO**

- **VUID-VkCommandBufferAllocateInfo-pNext-pNext**
  `pNext` must be `NULL`

- **VUID-VkCommandBufferAllocateInfo-commandPool-parameter**
  `commandPool` must be a valid `VkCommandPool` handle

- **VUID-VkCommandBufferAllocateInfo-level-parameter**
  `level` must be a valid `VkCommandBufferLevel` value
Possible values of `VkCommandBufferAllocateInfo::level`, specifying the command buffer level, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandBufferLevel {
    VK_COMMAND_BUFFER_LEVEL_PRIMARY = 0,
    VK_COMMAND_BUFFER_LEVEL_SECONDARY = 1,
} VkCommandBufferLevel;
```

- `VK_COMMAND_BUFFER_LEVEL_PRIMARY` specifies a primary command buffer.
- `VK_COMMAND_BUFFER_LEVEL_SECONDARY` specifies a secondary command buffer.

To reset command buffers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetCommandBuffer(
    VkCommandBuffer commandBuffer,
    VkCommandBufferResetFlags flags);
```

- `commandBuffer` is the command buffer to reset. The command buffer can be in any state other than pending, and is moved into the initial state.
- `flags` is a bitmask of `VkCommandBufferResetFlagBits` controlling the reset operation.

Any primary command buffer that is in the recording or executable state and has `commandBuffer` recorded into it, becomes invalid.

**Valid Usage**

- `VUID-vkResetCommandBuffer-commandBuffer-00045`
  - `commandBuffer` must not be in the pending state
- `VUID-vkResetCommandBuffer-commandBuffer-00046`
  - `commandBuffer` must have been allocated from a pool that was created with the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`

**Valid Usage (Implicit)**

- `VUID-vkResetCommandBuffer-commandBuffer-parameter`
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- `VUID-vkResetCommandBuffer-flags-parameter`
  - `flags` must be a valid combination of `VkCommandBufferResetFlagBits` values
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Return Codes

Success

- `VK_SUCCESS`

Failure

- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Bits which can be set in `vkResetCommandBuffer::flags` to control the reset operation are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandBufferResetFlagBits {
    VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
} VkCommandBufferResetFlagBits;
```

- `VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT` specifies that most or all memory resources currently owned by the command buffer should be returned to the parent command pool. If this flag is not set, then the command buffer may hold onto memory resources and reuse them when recording commands. `commandBuffer` is moved to the initial state.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandBufferResetFlags;
```

`VkCommandBufferResetFlags` is a bitmask type for setting a mask of zero or more `VkCommandBufferResetFlagBits`.

To free command buffers, call:

```c
// Provided by VK_VERSION_1_0
void vkFreeCommandBuffers(
    VkDevice device,
    VkCommandPool commandPool,
    uint32_t commandBufferCount,
    const VkCommandBuffer* pCommandBuffers);
```

- `device` is the logical device that owns the command pool.
- `commandPool` is the command pool from which the command buffers were allocated.
- `commandBufferCount` is the length of the `pCommandBuffers` array.
• *pCommandBuffers* is a pointer to an array of handles of command buffers to free.

Any primary command buffer that is in the recording or executable state and has any element of *pCommandBuffers* recorded into it, becomes invalid.

### Valid Usage

- VUID-vkFreeCommandBuffers-pCommandBuffers-00047
  All elements of *pCommandBuffers* must not be in the pending state

- VUID-vkFreeCommandBuffers-pCommandBuffers-00048
  *pCommandBuffers* must be a valid pointer to an array of `commandBufferCount` `VkCommandBuffer` handles, each element of which must either be a valid handle or NULL

### Valid Usage (Implicit)

- VUID-vkFreeCommandBuffers-device-parameter
  *device* must be a valid `VkDevice` handle

- VUID-vkFreeCommandBuffers-commandPool-parameter
  *commandPool* must be a valid `VkCommandPool` handle

- VUID-vkFreeCommandBuffers-commandBufferCount-arraylength
  *commandBufferCount* must be greater than 0

- VUID-vkFreeCommandBuffers-commandPool-parent
  *commandPool* must have been created, allocated, or retrieved from *device*

- VUID-vkFreeCommandBuffers-pCommandBuffers-parent
  Each element of *pCommandBuffers* that is a valid handle must have been created, allocated, or retrieved from *commandPool*

### Host Synchronization

- Host access to *commandPool* must be externally synchronized

- Host access to each member of *pCommandBuffers* must be externally synchronized

### 6.4. Command Buffer Recording

To begin recording a command buffer, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkBeginCommandBuffer(
    VkCommandBuffer                          commandBuffer,
    const VkCommandBufferBeginInfo*         pBeginInfo);
```
• **commandBuffer** is the handle of the command buffer which is to be put in the recording state.

• **pBeginInfo** points to a VkCommandBufferBeginInfo structure defining additional information about how the command buffer begins recording.

---

### Valid Usage

- **VUID-vkBeginCommandBuffer-commandBuffer-00049**
  
  **commandBuffer must not be in the recording or pending state**

- **VUID-vkBeginCommandBuffer-commandBuffer-00050**
  
  If commandBuffer was allocated from a VkCommandPool which did not have the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT flag set, commandBuffer must be in the initial state.

- **VUID-vkBeginCommandBuffer-commandBuffer-00051**
  
  If commandBuffer is a secondary command buffer, the pInheritanceInfo member of pBeginInfo must be a valid VkCommandBufferInheritanceInfo structure.

- **VUID-vkBeginCommandBuffer-commandBuffer-00052**
  
  If commandBuffer is a secondary command buffer and either the occlusionQueryEnable member of the pInheritanceInfo member of pBeginInfo is VK_FALSE, or the precise occlusion queries feature is not enabled, the queryFlags member of the pInheritanceInfo member pBeginInfo must not contain VK_QUERY_CONTROL_PRECISE_BIT.

- **VUID-vkBeginCommandBuffer-commandBuffer-02840**
  
  If commandBuffer is a primary command buffer, then pBeginInfo->flags must not set both the VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT and the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT flags.

---

### Valid Usage (Implicit)

- **VUID-vkBeginCommandBuffer-commandBuffer-parameter**
  
  commandBuffer must be a valid VkCommandBuffer handle.

- **VUID-vkBeginCommandBuffer-pBeginInfo-parameter**
  
  pBeginInfo must be a valid pointer to a valid VkCommandBufferBeginInfo structure.

---

### Host Synchronization

- Host access to commandBuffer must be externally synchronized.

- Host access to theVkCommandPool that commandBuffer was allocated from must be externally synchronized.
Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkCommandBufferBeginInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCommandBufferBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkCommandBufferUsageFlags flags;
    const VkCommandBufferInheritanceInfo* pInheritanceInfo;
} VkCommandBufferBeginInfo;
```

• `sType` is the type of this structure.

• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `flags` is a bitmask of `VkCommandBufferUsageFlagBits` specifying usage behavior for the command buffer.

• `pInheritanceInfo` is a pointer to a `VkCommandBufferInheritanceInfo` structure, used if `commandBuffer` is a secondary command buffer. If this is a primary command buffer, then this value is ignored.

Valid Usage

• VUID-VkCommandBufferBeginInfo-flags-00053

  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `renderPass` member of `pInheritanceInfo` must be a valid `VkRenderPass`

• VUID-VkCommandBufferBeginInfo-flags-00054

  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `subpass` member of `pInheritanceInfo` must be a valid subpass index within the `renderPass` member of `pInheritanceInfo`

• VUID-VkCommandBufferBeginInfo-flags-00055

  If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `framebuffer` member of `pInheritanceInfo` must be either `VK_NULL_HANDLE`, or a valid `VkFramebuffer` that is compatible with the `renderPass` member of `pInheritanceInfo`
Valid Usage (Implicit)

- **VUID-VkCommandBufferBeginInfo-sType-sType**
  *sType must be* `VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO`

- **VUID-VkCommandBufferBeginInfo-pNext-pNext**
  *pNext must be* `NULL` *or a pointer to a valid instance of* `VkDeviceGroupCommandBufferBeginInfo`

- **VUID-VkCommandBufferBeginInfo-sType-unique**
  The *sType* value of each struct in the *pNext* chain must be unique

- **VUID-VkCommandBufferBeginInfo-flags-parameter**
  *flags must be a valid combination of* `VkCommandBufferUsageFlagBits` *values*

Bits which can be set in `VkCommandBufferBeginInfo::flags` to specify usage behavior for a command buffer are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkCommandBufferUsageFlagBits {
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT = 0x00000001,
    VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT = 0x00000002,
    VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT = 0x00000004,
} VkCommandBufferUsageFlagBits;
```

- **VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT** specifies that each recording of the command buffer will only be submitted once, and the command buffer will be reset and recorded again between each submission.

- **VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT** specifies that a secondary command buffer is considered to be entirely inside a render pass. If this is a primary command buffer, then this bit is ignored.

- **VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT** specifies that a command buffer can be resubmitted to a queue while it is in the *pending state*, and recorded into multiple primary command buffers.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkCommandBufferUsageFlags;
```

*VkCommandBufferUsageFlags* is a bitmask type for setting a mask of zero or more `VkCommandBufferUsageFlagBits`.

If the command buffer is a secondary command buffer, then the `VkCommandBufferInheritanceInfo` structure defines any state that will be inherited from the primary command buffer:
// Provided by VK_VERSION_1_0
typedef struct VkCommandBufferInheritanceInfo {
    VkStructureType                  sType;
    const void*                      pNext;
    VkRenderPass                     renderPass;
    uint32_t                         subpass;
    VkFramebuffer                    framebuffer;
    VkBool32                         occlusionQueryEnable;
    VkQueryControlFlags              queryFlags;
    VkQueryPipelineStatisticFlags    pipelineStatistics;
} VkCommandBufferInheritanceInfo;

- **sType** is the type of this structure.

- **pNext** is NULL or a pointer to a structure extending this structure.

- **renderPass** is a VkRenderPass object defining which render passes the VkCommandBuffer will be compatible with and can be executed within. If the VkCommandBuffer will not be executed within a render pass instance, renderPass is ignored.

- **subpass** is the index of the subpass within the render pass instance that the VkCommandBuffer will be executed within. If the VkCommandBuffer will not be executed within a render pass instance, subpass is ignored.

- **framebuffer** optionally refers to the VkFramebuffer object that the VkCommandBuffer will be rendering to if it is executed within a render pass instance. It can be VK_NULL_HANDLE if the framebuffer is not known, or if the VkCommandBuffer will not be executed within a render pass instance.

  **Note**
  Specifying the exact framebuffer that the secondary command buffer will be executed with may result in better performance at command buffer execution time.

- **occlusionQueryEnable** specifies whether the command buffer can be executed while an occlusion query is active in the primary command buffer. If this is VK_TRUE, then this command buffer can be executed whether the primary command buffer has an occlusion query active or not. If this is VK_FALSE, then the primary command buffer must not have an occlusion query active.

- **queryFlags** specifies the query flags that can be used by an active occlusion query in the primary command buffer when this secondary command buffer is executed. If this value includes the VK_QUERY_CONTROL_PRECISE_BIT bit, then the active query can return boolean results or actual sample counts. If this bit is not set, then the active query must not use the VK_QUERY_CONTROL_PRECISE_BIT bit.

- **pipelineStatistics** is a bitmask of VkQueryPipelineStatisticFlagBits specifying the set of pipeline statistics that can be counted by an active query in the primary command buffer when this secondary command buffer is executed. If this value includes a given bit, then this command buffer can be executed whether the primary command buffer has a pipeline statistics query active that includes this bit or not. If this value excludes a given bit, then the
active pipeline statistics query must not be from a query pool that counts that statistic.

Valid Usage

- VUID-VkCommandBufferInheritanceInfo-occlusionQueryEnable-00056
  If the inherited queries feature is not enabled, occlusionQueryEnable must be VK_FALSE

- VUID-VkCommandBufferInheritanceInfo-queryFlags-00057
  If the inherited queries feature is enabled, queryFlags must be a valid combination of VkQueryControlFlagBits values

- VUID-VkCommandBufferInheritanceInfo-queryFlags-02788
  If the inherited queries feature is not enabled, queryFlags must be 0

- VUID-VkCommandBufferInheritanceInfo-pipelineStatistics-02789
  If the pipeline statistics queries feature is enabled, pipelineStatistics must be a valid combination of VkQueryPipelineStatisticFlagBits values

- VUID-VkCommandBufferInheritanceInfo-pipelineStatistics-00058
  If the pipeline statistics queries feature is not enabled, pipelineStatistics must be 0

Valid Usage (Implicit)

- VUID-VkCommandBufferInheritanceInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO

- VUID-VkCommandBufferInheritanceInfo-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkCommandBufferInheritanceConditionalRenderingInfoEXT, VkCommandBufferInheritanceRenderPassTransformInfoQCOM, or VkCommandBufferInheritanceViewportScissorInfoNV

- VUID-VkCommandBufferInheritanceInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkCommandBufferInheritanceInfo-commonparent
  Both of framebuffer, and renderPass that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

Note

On some implementations, not using the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT bit enables command buffers to be patched in-place if needed, rather than creating a copy of the command buffer.

If a command buffer is in the invalid, or executable state, and the command buffer was allocated from a command pool with the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT flag set, then vkBeginCommandBuffer implicitly resets the command buffer, behaving as if vkResetCommandBuffer had been called with VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT not set. After the implicit reset, commandBuffer is moved to the recording state.
If the `pNext` chain of `VkCommandBufferInheritanceInfo` includes a `VkCommandBufferInheritanceConditionalRenderingInfoEXT` structure, then that structure controls whether a command buffer can be executed while conditional rendering is active in the primary command buffer.

The `VkCommandBufferInheritanceConditionalRenderingInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_conditional_rendering
typedef struct VkCommandBufferInheritanceConditionalRenderingInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkBool32 conditionalRenderingEnable;
} VkCommandBufferInheritanceConditionalRenderingInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `conditionalRenderingEnable` specifies whether the command buffer can be executed while conditional rendering is active in the primary command buffer. If this is `VK_TRUE`, then this command buffer can be executed whether the primary command buffer has active conditional rendering or not. If this is `VK_FALSE`, then the primary command buffer must not have conditional rendering active.

If this structure is not present, the behavior is as if `conditionalRenderingEnable` is `VK_FALSE`.

### Valid Usage

- VUID-VkCommandBufferInheritanceConditionalRenderingInfoEXT-conditionalRenderingEnable-01977
  - If the inherited conditional rendering feature is not enabled, `conditionalRenderingEnable` must be `VK_FALSE`

### Valid Usage (Implicit)

- VUID-VkCommandBufferInheritanceConditionalRenderingInfoEXT-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_CONDITIONAL_RENDERING_INFO_EXT`

To begin recording a secondary command buffer compatible with execution inside a render pass using render pass transform, add the `VkCommandBufferInheritanceRenderPassTransformInfoQCOM` to the `pNext` chain of `VkCommandBufferInheritanceInfo` structure passed to the `vkBeginCommandBuffer` command specifying the parameters for transformed rasterization.

The `VkCommandBufferInheritanceRenderPassTransformInfoQCOM` structure is defined as:
typedef struct VkCommandBufferInheritanceRenderPassTransformInfoQCOM {
    VkStructureType sType;
    void* pNext;
    VkSurfaceTransformFlagBitsKHR transform;
    VkRect2D renderArea;
} VkCommandBufferInheritanceRenderPassTransformInfoQCOM;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **transform** is a VkSurfaceTransformFlagBitsKHR value describing the transform to be applied to the render pass.
- **renderArea** is the render area that is affected by the command buffer.

When the secondary is recorded to execute within a render pass instance using `vkCmdExecuteCommands`, the render pass transform parameters of the secondary command buffer must be consistent with the render pass transform parameters specified for the render pass instance. In particular, the **transform** and **renderArea** for command buffer must be identical to the **transform** and **renderArea** of the render pass instance.

### Valid Usage

- VUID-VkCommandBufferInheritanceRenderPassTransformInfoQCOM-transform-02864
  - **transform** must be: VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR, VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR, VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR, or VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR

### Valid Usage (Implicit)

- VUID-VkCommandBufferInheritanceRenderPassTransformInfoQCOM-sType-sType
  - **sType** must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_RENDER_PASS_TRANSFORM_INFO_QCOM

The **VkCommandBufferInheritanceViewportScissorInfoNV** structure is defined as:

// Provided by VK_NV_inherited_viewport_scissor
typedef struct VkCommandBufferInheritanceViewportScissorInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkBool32 viewportScissor2D;
    uint32_t viewportDepthCount;
    const VkViewport* pViewportDepths;
} VkCommandBufferInheritanceViewportScissorInfoNV;
• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• viewportScissor2D specifies whether the listed dynamic state is inherited.
• viewportDepthCount specifies the maximum number of viewports to inherit. When viewportScissor2D is VK_FALSE, the behavior is as if this value is zero.
• pViewportDepths specifies the expected depth range for each inherited viewport.

If the pNext chain of VkCommandBufferInheritanceInfo includes a VkCommandBufferInheritanceViewportScissorInfoNV structure, then that structure controls whether a command buffer can inherit the following state from other command buffers:

• VK_DYNAMIC_STATE_SCISSOR
• VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT
• VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT

as well as the following state, with restrictions on inherited depth values and viewport count:

• VK_DYNAMIC_STATE_VIEWPORT
• VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT

If viewportScissor2D is VK_FALSE, then the command buffer does not inherit the listed dynamic state, and should set this state itself. If this structure is not present, the behavior is as if viewportScissor2D is VK_FALSE.

If viewportScissor2D is VK_TRUE, then the listed dynamic state is inherited, and the command buffer must not set this state, except that the viewport and scissor count may be set by binding a graphics pipeline that does not specify this state as dynamic.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due to this restriction, applications should ensure either all or none of the graphics pipelines bound in this secondary command buffer use dynamic viewport/scissor counts.</td>
</tr>
</tbody>
</table>

When the command buffer is executed as part of the execution of a vkCmdExecuteCommands command, the inherited state (if enabled) is determined by the following procedure, performed separately for each dynamic state, and separately for each value for dynamic state that consists of multiple values (e.g. multiple viewports).

• With \( i \) being the index of the executed command buffer in the pCommandBuffers array of vkCmdExecuteCommands, if \( i > 0 \) and any secondary command buffer from index 0 to \( i - 1 \) modifies the state, the inherited state is provisionally set to the final value set by the last such secondary command buffer. Binding a graphics pipeline that defines the state statically is equivalent to setting the state to an undefined value.

• Otherwise, the tentatative inherited state is that of the primary command buffer at the point the vkCmdExecuteCommands command was recorded; if the state is undefined, then so is the provisional inherited state.
• If the provisional inherited state is an undefined value, then the state is not inherited.
• If the provisional inherited state is a viewport, with \( n \) being its viewport index, then if \( n \geq \text{viewportDepthCount} \), or if either \( \text{VkViewport::minDepth} \) or \( \text{VkViewport::maxDepth} \) are not equal to the respective values of the \( n \)th element of \( p\text{ViewDepths} \), then the state is not inherited.
• If the provisional inherited state passes both checks, then it becomes the actual inherited state.

Note
There is no support for inheriting dynamic state from a secondary command buffer executed as part of a different \( \text{vkCmdExecuteCommands} \) command.

Valid Usage

- VUID-VkCommandBufferInheritanceViewportScissorInfoNV-viewportScissor2D-04782
  If the inherited viewport scissor feature is not enabled, \text{viewportScissor2D} must be \text{VK_FALSE}.

- VUID-VkCommandBufferInheritanceViewportScissorInfoNV-viewportScissor2D-04783
  If the multiple viewports feature is not enabled and \text{viewportScissor2D} is \text{VK_TRUE}, then \text{viewportDepthCount} must be \text{1}.

- VUID-VkCommandBufferInheritanceViewportScissorInfoNV-viewportScissor2D-04784
  If \text{viewportScissor2D} is \text{VK_TRUE}, then \text{viewportDepthCount} must be greater than \text{0}.

- VUID-VkCommandBufferInheritanceViewportScissorInfoNV-viewportScissor2D-04785
  If \text{viewportScissor2D} is \text{VK_TRUE}, then \( p\text{ViewportDepths} \) must be a valid pointer to an array of \text{viewportDepthCount} valid \text{VkViewport} structures, except any requirements on \( x \), \( y \), \( width \), and \( height \) do not apply.

- VUID-VkCommandBufferInheritanceViewportScissorInfoNV-viewportScissor2D-04786
  If \text{viewportScissor2D} is \text{VK_TRUE}, then the command buffer must be recorded with the \text{VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT}.

Valid Usage (Implicit)
- VUID-VkCommandBufferInheritanceViewportScissorInfoNV-sType-sType
  \text{sType} must be \text{VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_VIEWPORT_SCISSOR_INFO_NV}.

Once recording starts, an application records a sequence of commands (\( \text{vkCmd*} \)) to set state in the command buffer, draw, dispatch, and other commands.

Several commands can also be recorded indirectly from \text{VkBuffer} content, see Device-Generated Commands.

To complete recording of a command buffer, call:
// Provided by VK_VERSION_1_0
VkResult vkEndCommandBuffer(
    VkCommandBuffer commandBuffer);

- `commandBuffer` is the command buffer to complete recording.

If there was an error during recording, the application will be notified by an unsuccessful return code returned by `vkEndCommandBuffer`. If the application wishes to further use the command buffer, the command buffer **must** be reset.

The command buffer **must** have been in the **recording state**, and is moved to the **executable state**.

### Valid Usage

- VUID-vkEndCommandBuffer-commandBuffer-00059
  ```
  commandBuffer must be in the recording state
  ```

- VUID-vkEndCommandBuffer-commandBuffer-00060
  ```
  If commandBuffer is a primary command buffer, there must not be an active render pass instance
  ```

- VUID-vkEndCommandBuffer-commandBuffer-00061
  ```
  All queries made active during the recording of commandBuffer must have been made inactive
  ```

- VUID-vkEndCommandBuffer-None-01978
  ```
  Conditional rendering must not be active
  ```

- VUID-vkEndCommandBuffer-commandBuffer-01815
  ```
  If commandBuffer is a secondary command buffer, there must not be an outstanding vkCmdBeginDebugUtilsLabelEXT command recorded to commandBuffer that has not previously been ended by a call to vkCmdEndDebugUtilsLabelEXT
  ```

- VUID-vkEndCommandBuffer-commandBuffer-00062
  ```
  If commandBuffer is a secondary command buffer, there must not be an outstanding vkCmdDebugMarkerBeginEXT command recorded to commandBuffer that has not previously been ended by a call to vkCmdDebugMarkerEndEXT
  ```

### Valid Usage (Implicit)

- VUID-vkEndCommandBuffer-commandBuffer-parameter
  ```
  commandBuffer must be a valid VkCommandBuffer handle
  ```
Host Synchronization

• Host access to commandBuffer must be externally synchronized
• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

When a command buffer is in the executable state, it can be submitted to a queue for execution.

6.5. Command Buffer Submission

Note
Submission can be a high overhead operation, and applications should attempt to batch work together into as few calls to vkQueueSubmit or vkQueueSubmit2KHR as possible.

To submit command buffers to a queue, call:

```c
// Provided by VK_KHR_synchronization2
VkResult vkQueueSubmit2KHR(
    VkQueue queue,
    uint32_t submitCount,
    const VkSubmitInfo2KHR* pSubmits,
    VkFence fence);
```

• queue is the queue that the command buffers will be submitted to.
• submitCount is the number of elements in the pSubmits array.
• pSubmits is a pointer to an array of VkSubmitInfo2KHR structures, each specifying a command buffer submission batch.
• fence is an optional handle to a fence to be signaled once all submitted command buffers have completed execution. If fence is not VK_NULL_HANDLE, it defines a fence signal operation.

vkQueueSubmit2KHR is a queue submission command, with each batch defined by an element of pSubmits as an instance of the VkSubmitInfo2KHR structure.
Semaphore operations submitted with `vkQueueSubmit2KHR` have additional ordering constraints compared to other submission commands, with dependencies involving previous and subsequent queue operations. Information about these additional constraints can be found in the semaphore section of the synchronization chapter.

If any command buffer submitted to this queue is in the executable state, it is moved to the pending state. Once execution of all submissions of a command buffer complete, it moves from the pending state, back to the executable state. If a command buffer was recorded with the `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` flag, it instead moves back to the invalid state.

If `vkQueueSubmit2KHR` fails, it may return `VK_ERROR_OUT_OF_HOST_MEMORY` or `VK_ERROR_OUT_OF_DEVICE_MEMORY`. If it does, the implementation must ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by `pSubmits` is unaffected by the call or its failure. If `vkQueueSubmit2KHR` fails in such a way that the implementation is unable to make that guarantee, the implementation must return `VK_ERROR_DEVICE_LOST`. See Lost Device.
Valid Usage

- **VUID-vkQueueSubmit2KHR-synchronization2-03866**
  The synchronization2 feature must be enabled

- **VUID-vkQueueSubmit2KHR-commandBuffer-03867**
  If a command recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits referenced an VkEvent, that event must not be referenced by a command that has been submitted to another queue and is still in the pending state

- **VUID-vkQueueSubmit2KHR-semaphore-03868**
  The semaphore member of any element of the pSignalSemaphoreInfos member of any element of pSubmits must be unsignaled when the semaphore signal operation it defines is executed on the device

- **VUID-vkQueueSubmit2KHR-stageMask-03869**
  The stageMask member of any element of the pSignalSemaphoreInfos member of any element of pSubmits must only include pipeline stages that are supported by the queue family which queue belongs to

- **VUID-vkQueueSubmit2KHR-stageMask-03870**
  The stageMask member of any element of the pWaitSemaphoreInfos member of any element of pSubmits must only include pipeline stages that are supported by the queue family which queue belongs to

- **VUID-vkQueueSubmit2KHR-semaphore-03871**
  When a semaphore wait operation for a binary semaphore is executed, as defined by the semaphore member of any element of the pWaitSemaphoreInfos member of any element of pSubmits, there must be no other queues waiting on the same semaphore

- **VUID-vkQueueSubmit2KHR-semaphore-03872**
  The semaphore member of any element of the pWaitSemaphoreInfos member of any element of pSubmits must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution

- **VUID-vkQueueSubmit2KHR-semaphore-03873**
  Any semaphore member of any element of the pWaitSemaphoreInfos member of any element of pSubmits that was created with a VkSemaphoreTypeKHR of VK_SEMAPHORE_TYPE_BINARY_KHR must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution

- **VUID-vkQueueSubmit2KHR-commandBuffer-03874**
  The commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits must be in the pending or executable state

- **VUID-vkQueueSubmit2KHR-commandBuffer-03875**
  If a command recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- **VUID-vkQueueSubmit2KHR-commandBuffer-03876**
Any secondary command buffers recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits must be in the pending or executable state

- VUID-vkQueueSubmit2KHR-commandBuffer-03877
  If any secondary command buffers recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- VUID-vkQueueSubmit2KHR-commandBuffer-03878
  The commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits must have been allocated from a VkCommandPool that was created for the same queue family queue belongs to

- VUID-vkQueueSubmit2KHR-commandBuffer-03879
  If a command recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits includes a Queue Family Transfer Acquire Operation, there must exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens before the acquire operation

- VUID-vkQueueSubmit2KHR-commandBuffer-03880
  If a command recorded into the commandBuffer member of any element of the pCommandBufferInfos member of any element of pSubmits was a vkCmdBeginQuery whose queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the profiling lock must have been held continuously on the VkDevice that queue was retrieved from, throughout recording of those command buffers

---

**Valid Usage (Implicit)**

- VUID-vkQueueSubmit2KHR-queue-parameter
  queue must be a valid VkQueue handle

- VUID-vkQueueSubmit2KHR-pSubmits-parameter
  If submitCount is not 0, pSubmits must be a valid pointer to an array of submitCount valid VkSubmitInfo2KHR structures

- VUID-vkQueueSubmit2KHR-fence-parameter
  If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle

- VUID-vkQueueSubmit2KHR-commonparent
  Both of fence, and queue that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice
Host Synchronization

- Host access to queue must be externally synchronized
- Host access to fence must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

The VkSubmitInfo2KHR structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkSubmitInfo2KHR {
    VkStructureType                        sType;
    const void *                            pNext;
    VkSubmitFlagsKHR                       flags;
    uint32_t                               waitSemaphoreInfoCount;
    const VkSemaphoreSubmitInfoKHR*        pWaitSemaphoreInfos;
    uint32_t                               commandBufferInfoCount;
    const VkCommandBufferSubmitInfoKHR*    pCommandBufferInfos;
    uint32_t                               signalSemaphoreInfoCount;
    const VkSemaphoreSubmitInfoKHR*        pSignalSemaphoreInfos;
} VkSubmitInfo2KHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask of VkSubmitFlagBitsKHR.
- waitSemaphoreInfoCount is the number of elements in pWaitSemaphoreInfos.
- pWaitSemaphoreInfos is a pointer to an array of VkSemaphoreSubmitInfoKHR structures defining
semaphore wait operations.

- `commandBufferInfoCount` is the number of elements in `pCommandBufferInfos` and the number of command buffers to execute in the batch.
- `pCommandBufferInfos` is a pointer to an array of `VkCommandBufferSubmitInfoKHR` structures describing command buffers to execute in the batch.
- `signalSemaphoreInfoCount` is the number of elements in `pSignalSemaphoreInfos`.
- `pSignalSemaphoreInfos` is a pointer to an array of `VkSemaphoreSubmitInfoKHR` describing semaphore signal operations.

### Valid Usage

- **VUID-VkSubmitInfo2KHR-semaphore-03881**
  
  If the same semaphore is used as the `semaphore` member of both an element of `pSignalSemaphoreInfos` and `pWaitSemaphoreInfos`, and that semaphore is a timeline semaphore, the `value` member of the `pSignalSemaphoreInfos` element **must** be greater than the `value` member of the `pWaitSemaphoreInfos` element.

- **VUID-VkSubmitInfo2KHR-semaphore-03882**
  
  If the `semaphore` member of any element of `pSignalSemaphoreInfos` is a timeline semaphore, the `value` member of that element **must** have a value greater than the current value of the semaphore when the semaphore signal operation is executed.

- **VUID-VkSubmitInfo2KHR-semaphore-03883**
  
  If the `semaphore` member of any element of `pSignalSemaphoreInfos` is a timeline semaphore, the `value` member of that element **must** have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

- **VUID-VkSubmitInfo2KHR-semaphore-03884**
  
  If the `semaphore` member of any element of `pWaitSemaphoreInfos` is a timeline semaphore, the `value` member of that element **must** have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

- **VUID-VkSubmitInfo2KHR-flags-03885**
  
  If the protected memory feature is not enabled, `flags` **must** not include `VK_SUBMIT_PROTECTED_BIT_KHR`.

- **VUID-VkSubmitInfo2KHR-flags-03886**
  
  If `flags` includes `VK_SUBMIT_PROTECTED_BIT_KHR`, all elements of `pCommandBuffers` **must** be protected command buffers.

- **VUID-VkSubmitInfo2KHR-flags-03887**
  
  If `flags` does not include `VK_SUBMIT_PROTECTED_BIT_KHR`, each element of `pCommandBuffers` **must** not be a protected command buffer.
Valid Usage (Implicit)

- VUID-VkSubmitInfo2KHR-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_SUBMIT_INFO_2_KHR`.

- VUID-VkSubmitInfo2KHR-pNext-pNext
  
  Each **pNext** member of any structure (including this one) in the **pNext** chain must be either NULL or a pointer to a valid instance of `VkPerformanceQuerySubmitInfoKHR`, `VkWin32KeyedMutexAcquireReleaseInfoKHR`, or `VkWin32KeyedMutexAcquireReleaseInfoNV`.

- VUID-VkSubmitInfo2KHR-sType-unique
  
  The **sType** value of each struct in the **pNext** chain must be unique.

- VUID-VkSubmitInfo2KHR-flags-parameter
  
  **flags** must be a valid combination of `VkSubmitFlagBitsKHR` values.

- VUID-VkSubmitInfo2KHR-pWaitSemaphoreInfos-parameter
  
  If `waitSemaphoreInfoCount` is not 0, *pWaitSemaphoreInfos* must be a valid pointer to an array of `waitSemaphoreInfoCount` valid `VkSemaphoreSubmitInfoKHR` structures.

- VUID-VkSubmitInfo2KHR-pCommandBufferInfos-parameter
  
  If `commandBufferInfoCount` is not 0, *pCommandBufferInfos* must be a valid pointer to an array of `commandBufferInfoCount` valid `VkCommandBufferSubmitInfoKHR` structures.

- VUID-VkSubmitInfo2KHR-pSignalSemaphoreInfos-parameter
  
  If `signalSemaphoreInfoCount` is not 0, *pSignalSemaphoreInfos* must be a valid pointer to an array of `signalSemaphoreInfoCount` valid `VkSemaphoreSubmitInfoKHR` structures.

Bits which can be set in `VkSubmitInfo2KHR::flags` to specify submission behavior are:

```c
// Provided by VK_KHR_synchronization2
typedef enum VkSubmitFlagBitsKHR {
  VK_SUBMIT_PROTECTED_BIT_KHR = 0x00000001,
} VkSubmitFlagBitsKHR;
```

- **VK_SUBMIT_PROTECTED_BIT_KHR** specifies that this batch is a protected submission.

```c
// Provided by VK_KHR_synchronization2
typedef VkFlags VkSubmitFlagsKHR;
```

`VkSubmitFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSubmitFlagBitsKHR`.

The `VkSemaphoreSubmitInfoKHR` structure is defined as:
typedef struct VkSemaphoreSubmitInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    uint64_t value;
    VkPipelineStageFlags2KHR stageMask;
    uint32_t deviceIndex;
} VkSemaphoreSubmitInfoKHR;

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `semaphore` is a `VkSemaphore` affected by this operation.
- `value` is either the value used to signal `semaphore` or the value waited on by `semaphore`, if `semaphore` is a timeline semaphore. Otherwise it is ignored.
- `stageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages at which the semaphore either waits or is signaled.
- `deviceIndex` is the index of the device within a device group that executes the semaphore wait or signal operation.

Whether this structure defines a semaphore wait or signal operation is defined by how it is used.
Valid Usage

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03929
  If the **geometry shaders** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03930
  If the **tessellation shaders** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_TESSellation_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSellation_EVALUATION_SHADER_BIT_KHR`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03931
  If the **conditional rendering** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03932
  If the **fragment density map** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03933
  If the **transform feedback** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03934
  If the **mesh shaders** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03935
  If the **task shaders** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-03936
  If the **shading rate image** feature is not enabled, **stageMask** **must** not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`

- VUID-VkSemaphoreSubmitInfoKHR-device-03888
  If the **device that semaphore was created on** is not a device group, **deviceIndex** **must** be 0

- VUID-VkSemaphoreSubmitInfoKHR-device-03889
  If the **device that semaphore was created on** is a device group, **deviceIndex** **must** be a valid device index

Valid Usage (Implicit)

- VUID-VkSemaphoreSubmitInfoKHR-sType-sType
  **sType** **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_SUBMIT_INFO_KHR`

- VUID-VkSemaphoreSubmitInfoKHR-pNext-pNext
  **pNext** **must** be `NULL`

- VUID-VkSemaphoreSubmitInfoKHR-semaphore-parameter
  **semaphore** **must** be a valid `VkSemaphore` handle

- VUID-VkSemaphoreSubmitInfoKHR-stageMask-parameter
  **stageMask** **must** be a valid combination of `VkPipelineStageFlagBits2KHR` values
The `VkCommandBufferSubmitInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkCommandBufferSubmitInfoKHR {
    VkStructureType   sType;
    const void*       pNext;
    VkCommandBuffer    commandBuffer;
    uint32_t           deviceMask;
} VkCommandBufferSubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `commandBuffer` is a `VkCommandBuffer` to be submitted for execution.
- `deviceMask` is a bitmask indicating which devices in a device group execute the command buffer. A `deviceMask` of `0` is equivalent to setting all bits corresponding to valid devices in the group to `1`.

## Valid Usage

- VUID-VkCommandBufferSubmitInfoKHR-commandBuffer-03890
  
  `commandBuffer` must not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`

- VUID-VkCommandBufferSubmitInfoKHR-deviceMask-03891
  
  If `deviceMask` is not `0`, it must be a valid device mask

## Valid Usage (Implicit)

- VUID-VkCommandBufferSubmitInfoKHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO_KHR`

- VUID-VkCommandBufferSubmitInfoKHR-pNext-pNext
  
  `pNext` must be `NULL`

- VUID-VkCommandBufferSubmitInfoKHR-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

To submit command buffers to a queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueSubmit(
    VkQueue                                     queue,
    uint32_t                                    submitCount,
    const VkSubmitInfo*                         pSubmits,
    VkFence                                     fence);
```

- `queue` is the queue that the command buffers will be submitted to.
• **submitCount** is the number of elements in the **pSubmits** array.

• **pSubmits** is a pointer to an array of **VkSubmitInfo** structures, each specifying a command buffer submission batch.

• **fence** is an **optional** handle to a fence to be signaled once all submitted command buffers have completed execution. If **fence** is not **VK_NULL_HANDLE**, it defines a **fence signal operation**.

**vkQueueSubmit** is a **queue submission command**, with each batch defined by an element of **pSubmits**. Batches begin execution in the order they appear in **pSubmits**, but **may** complete out of order.

Fence and semaphore operations submitted with **vkQueueSubmit** have additional ordering constraints compared to other submission commands, with dependencies involving previous and subsequent queue operations. Information about these additional constraints can be found in the **semaphore** and **fence** sections of the synchronization chapter.

Details on the interaction of **pWaitDstStageMask** with synchronization are described in the **semaphore wait operation** section of the synchronization chapter.

The order that batches appear in **pSubmits** is used to determine **submission order**, and thus all the **implicit ordering guarantees** that respect it. Other than these implicit ordering guarantees and any **explicit synchronization primitives**, these batches **may** overlap or otherwise execute out of order.

If any command buffer submitted to this queue is in the **executable state**, it is moved to the **pending state**. Once execution of all submissions of a command buffer complete, it moves from the **pending state**, back to the **executable state**. If a command buffer was recorded with the **VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT** flag, it instead moves to the **invalid state**.

If **vkQueueSubmit** fails, it **may** return **VK_ERROR_OUT_OF_HOST_MEMORY** or **VK_ERROR_OUT_OF_DEVICE_MEMORY**. If it does, the implementation **must** ensure that the state and contents of any resources or synchronization primitives referenced by the submitted command buffers and any semaphores referenced by **pSubmits** is unaffected by the call or its failure. If **vkQueueSubmit** fails in such a way that the implementation is unable to make that guarantee, the implementation **must** return **VK_ERROR_DEVICE_LOST**. See **Lost Device**.
Valid Usage

- VUID-vkQueueSubmit-fence-00063
  If fence is not VK_NULL_HANDLE, fence must be unsignaled

- VUID-vkQueueSubmit-fence-00064
  If fence is not VK_NULL_HANDLE, fence must not be associated with any other queue command that has not yet completed execution on that queue

- VUID-vkQueueSubmit-pCommandBuffers-00065
  Any calls to vkCmdSetEvent, vkCmdResetEvent or vkCmdWaitEvents that have been recorded into any of the command buffer elements of the pCommandBuffers member of any element of pSubmits, must not reference any VkEvent that is referenced by any of those commands in a command buffer that has been submitted to another queue and is still in the pending state

- VUID-vkQueueSubmit-pWaitDstStageMask-00066
  Any stage flag included in any element of the pWaitDstStageMask member of any element of pSubmits must be a pipeline stage supported by one of the capabilities of queue, as specified in the table of supported pipeline stages

- VUID-vkQueueSubmit-pSignalSemaphores-00067
  Each element of the pSignalSemaphores member of any element of pSubmits must be unsignaled when the semaphore signal operation it defines is executed on the device

- VUID-vkQueueSubmit-pWaitSemaphores-00068
  When a semaphore wait operation referring to a binary semaphore defined by any element of the pWaitSemaphores member of any element of pSubmits executes on queue, there must be no other queues waiting on the same semaphore

- VUID-vkQueueSubmit-pWaitSemaphores-03238
  All elements of the pWaitSemaphores member of all elements of pSubmits created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution

- VUID-vkQueueSubmit-pCommandBuffers-00070
  Each element of the pCommandBuffers member of each element of pSubmits must be in the pending or executable state

- VUID-vkQueueSubmit-pCommandBuffers-00071
  If any element of the pCommandBuffers member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state

- VUID-vkQueueSubmit-pCommandBuffers-00072
  Any secondary command buffers recorded into any element of the pCommandBuffers member of any element of pSubmits must be in the pending or executable state

- VUID-vkQueueSubmit-pCommandBuffers-00073
  If any secondary command buffers recorded into any element of the pCommandBuffers member of any element of pSubmits was not recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, it must not be in the pending state
Each element of the \texttt{pCommandBuffers} member of each element of \texttt{pSubmits} must have been allocated from a \texttt{VkCommandPool} that was created for the same queue family \texttt{queue} belongs to.

If any element of \texttt{pSubmits->pCommandBuffers} includes a Queue Family Transfer Acquire Operation, there must exist a previously submitted Queue Family Transfer Release Operation on a queue in the queue family identified by the acquire operation, with parameters matching the acquire operation as defined in the definition of such acquire operations, and which happens-before the acquire operation.

If a command recorded into any element of \texttt{pCommandBuffers} was a \texttt{vkCmdBeginQuery} whose queryPool was created with a queryType of \texttt{VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR}, the profiling lock must have been held continuously on the \texttt{VkDevice} that \texttt{queue} was retrieved from, throughout recording of those command buffers.

Any resource created with \texttt{VK_SHARING_MODE_EXCLUSIVE} that is read by an operation specified by \texttt{pSubmits} must not be owned by any queue family other than the one which \texttt{queue} belongs to, at the time it is executed.

Any resource created with \texttt{VK_SHARING_MODE_CONCURRENT} that is accessed by an operation specified by \texttt{pSubmits} must have included the queue family of \texttt{queue} at resource creation time.

### Valid Usage (Implicit)

- \texttt{VUID-vkQueueSubmit-queue-parameter} \\
  \texttt{queue} must be a valid \texttt{VkQueue} handle

- \texttt{VUID-vkQueueSubmit-pSubmits-parameter} \\
  If \texttt{submitCount} is not 0, \texttt{pSubmits} must be a valid pointer to an array of \texttt{submitCount} valid \texttt{VkSubmitInfo} structures

- \texttt{VUID-vkQueueSubmit-fence-parameter} \\
  If \texttt{fence} is not \texttt{VK_NULL_HANDLE}, \texttt{fence} must be a valid \texttt{VkFence} handle

- \texttt{VUID-vkQueueSubmit-commonparent} \\
  Both of \texttt{fence}, and \texttt{queue} that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same \texttt{VkDevice}

### Host Synchronization

- Host access to \texttt{queue} must be externally synchronized
- Host access to \texttt{fence} must be externally synchronized
Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>

Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

The `VkSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    const VkPipelineStageFlags* pWaitDstStageMask;
    uint32_t commandBufferCount;
    const VkCommandBuffer* pCommandBuffers;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkSubmitInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `waitSemaphoreCount` is the number of semaphores upon which to wait before executing the command buffers for the batch.
- `pWaitSemaphores` is a pointer to an array of `VkSemaphore` handles upon which to wait before the command buffers for this batch begin execution. If semaphores to wait on are provided, they define a semaphore wait operation.
- `pWaitDstStageMask` is a pointer to an array of pipeline stages at which each corresponding semaphore wait will occur.
- `commandBufferCount` is the number of command buffers to execute in the batch.
• **pCommandBuffers** is a pointer to an array of VkCommandBuffer handles to execute in the batch.

• **signalSemaphoreCount** is the number of semaphores to be signaled once the commands specified in pCommandBuffers have completed execution.

• **pSignalSemaphores** is a pointer to an array of VkSemaphore handles which will be signaled when the command buffers for this batch have completed execution. If semaphores to be signaled are provided, they define a semaphore signal operation.

The order that command buffers appear in **pCommandBuffers** is used to determine submission order, and thus all the implicit ordering guarantees that respect it. Other than these implicit ordering guarantees and any explicit synchronization primitives, these command buffers **may** overlap or otherwise execute out of order.
Valid Usage

- VUID-VkSubmitInfo-pCommandBuffers-00075
  Each element of `pCommandBuffers` must not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`

- VUID-VkSubmitInfo-pWaitDstStageMask-00076
  If the geometry shader feature is not enabled, each element of `pWaitDstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- VUID-VkSubmitInfo-pWaitDstStageMask-00077
  If the tessellation shader feature is not enabled, each element of `pWaitDstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- VUID-VkSubmitInfo-pWaitDstStageMask-00078
  Each element of `pWaitDstStageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`

- VUID-VkSubmitInfo-pWaitSemaphores-03239
  If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then the `pNext` chain must include a `VkTimelineSemaphoreSubmitInfo` structure

- VUID-VkSubmitInfo-pNext-03240
  If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pWaitSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then its `waitSemaphoreValueCount` member must equal `waitSemaphoreCount`

- VUID-VkSubmitInfo-pNext-03241
  If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, then its `signalSemaphoreValueCount` member must equal `signalSemaphoreCount`

- VUID-VkSubmitInfo-pSignalSemaphores-03242
  For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value greater than the current value of the semaphore when the semaphore signal operation is executed

- VUID-VkSubmitInfo-pWaitSemaphores-03243
  For each element of `pWaitSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pWaitSemaphoreValues` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`

- VUID-VkSubmitInfo-pSignalSemaphores-03244
  For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value which does
not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on that semaphore by more than maxTimelineSemaphoreValueDifference

- VUID-VkSubmitInfo-pWaitDstStageMask-02089
  If the mesh shaders feature is not enabled, each element of pWaitDstStageMask must not contain VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV

- VUID-VkSubmitInfo-pWaitDstStageMask-02090
  If the task shaders feature is not enabled, each element of pWaitDstStageMask must not contain VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV

- VUID-VkSubmitInfo-pNext-04120
  If the pNext chain of this structure does not include a VkProtectedSubmitInfo structure with protectedSubmit set to VK_TRUE, then each element of the pCommandBuffers array must be an unprotected command buffer

- VUID-VkSubmitInfo-pNext-04148
  If the pNext chain of this structure includes a VkProtectedSubmitInfo structure with protectedSubmit set to VK_TRUE, then each element of the pCommandBuffers array must be a protected command buffer
Valid Usage (Implicit)

- **VUID-VkSubmitInfo-sType-sType**
  
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkSubmitInfo-sType-unique**
  
  The `sType` must be `VK_STRUCTURE_TYPE_SUBMIT_INFO`.

- **VUID-VkSubmitInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkD3D12FenceSubmitInfoKHR`, `VkDeviceGroupSubmitInfo`, `VkPerformanceQuerySubmitInfoKHR`, `VkProtectedSubmitInfo`, `VkTimelineSemaphoreSubmitInfo`, `VkWin32KeyedMutexAcquireReleaseInfoKHR`, or `VkWin32KeyedMutexAcquireReleaseInfoNV`.

- **VUID-VkSubmitInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkD3D12FenceSubmitInfoKHR`, `VkDeviceGroupSubmitInfo`, `VkPerformanceQuerySubmitInfoKHR`, `VkProtectedSubmitInfo`, `VkTimelineSemaphoreSubmitInfo`, `VkWin32KeyedMutexAcquireReleaseInfoKHR`, or `VkWin32KeyedMutexAcquireReleaseInfoNV`.

To specify the values to use when waiting for and signaling semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, add a `VkTimelineSemaphoreSubmitInfo` structure to the `pNext` chain of the `VkSubmitInfo` structure when using `vkQueueSubmit` or the `VkBindSparseInfo` structure when using `vkQueueBindSparse`. The `VkTimelineSemaphoreSubmitInfo` structure is defined as:
```c
typedef struct VkTimelineSemaphoreSubmitInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           waitSemaphoreValueCount;
    const uint64_t*    pWaitSemaphoreValues;
    uint32_t           signalSemaphoreValueCount;
    const uint64_t*    pSignalSemaphoreValues;
} VkTimelineSemaphoreSubmitInfo;
```

or the equivalent

```c
// Provided by VK_KHR_timeline_semaphore
typedef VkTimelineSemaphoreSubmitInfo VkTimelineSemaphoreSubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `waitSemaphoreValueCount` is the number of semaphore wait values specified in `pWaitSemaphoreValues`.
- `pWaitSemaphoreValues` is an array of length `waitSemaphoreValueCount` containing values for the corresponding semaphores in `VkSubmitInfo::pWaitSemaphores` to wait for.
- `signalSemaphoreValueCount` is the number of semaphore signal values specified in `pSignalSemaphoreValues`.
- `pSignalSemaphoreValues` is an array of length `signalSemaphoreValueCount` containing values for the corresponding semaphores in `VkSubmitInfo::pSignalSemaphores` to set when signaled.

If the semaphore in `VkSubmitInfo::pWaitSemaphores` or `VkSubmitInfo::pSignalSemaphores` corresponding to an entry in `pWaitSemaphoreValues` or `pSignalSemaphoreValues` respectively was not created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, the implementation must ignore the value in the `pWaitSemaphoreValues` or `pSignalSemaphoreValues` entry.

**Valid Usage (Implicit)**

- VUID-VkTimelineSemaphoreSubmitInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO`
- VUID-VkTimelineSemaphoreSubmitInfo-pWaitSemaphoreValues-parameter
  If `waitSemaphoreValueCount` is not 0, and `pWaitSemaphoreValues` is not `NULL`, `pWaitSemaphoreValues` must be a valid pointer to an array of `waitSemaphoreValueCount` `uint64_t` values
- VUID-VkTimelineSemaphoreSubmitInfo-pSignalSemaphoreValues-parameter
  If `signalSemaphoreValueCount` is not 0, and `pSignalSemaphoreValues` is not `NULL`, `pSignalSemaphoreValues` must be a valid pointer to an array of `signalSemaphoreValueCount` `uint64_t` values
To specify the values to use when waiting for and signaling semaphores whose current payload refers to a Direct3D 12 fence, add a `VkD3D12FenceSubmitInfoKHR` structure to the `pNext` chain of the `VkSubmitInfo` structure. The `VkD3D12FenceSubmitInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_win32
typedef struct VkD3D12FenceSubmitInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           waitSemaphoreValuesCount;
    const uint64_t*    pWaitSemaphoreValues;
    uint32_t           signalSemaphoreValuesCount;
    const uint64_t*    pSignalSemaphoreValues;
} VkD3D12FenceSubmitInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `waitSemaphoreValuesCount` is the number of semaphore wait values specified in `pWaitSemaphoreValues`.
- `pWaitSemaphoreValues` is a pointer to an array of `waitSemaphoreValuesCount` values for the corresponding semaphores in `VkSubmitInfo::pWaitSemaphores` to wait for.
- `signalSemaphoreValuesCount` is the number of semaphore signal values specified in `pSignalSemaphoreValues`.
- `pSignalSemaphoreValues` is a pointer to an array of `signalSemaphoreValuesCount` values for the corresponding semaphores in `VkSubmitInfo::pSignalSemaphores` to set when signaled.

If the semaphore in `VkSubmitInfo::pWaitSemaphores` or `VkSubmitInfo::pSignalSemaphores` corresponding to an entry in `pWaitSemaphoreValues` or `pSignalSemaphoreValues` respectively does not currently have a payload referring to a Direct3D 12 fence, the implementation must ignore the value in the `pWaitSemaphoreValues` or `pSignalSemaphoreValues` entry.

**Note**
As the introduction of the external semaphore handle type `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT` predates that of timeline semaphores, support for importing semaphore payloads from external handles of that type into semaphores created (implicitly or explicitly) with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` is preserved for backwards compatibility. However, applications should prefer importing such handle types into semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`, and use the `VkTimelineSemaphoreSubmitInfo` structure instead of the `VkD3D12FenceSubmitInfoKHR` structure to specify the values to use when waiting for and signaling such semaphores.
Valid Usage

- VUID-VkD3D12FenceSubmitInfoKHR-waitSemaphoreValuesCount-00079
  \textit{waitSemaphoreValuesCount} \textbf{must} be the same value as \texttt{VkSubmitInfo::waitSemaphoreCount}, where \texttt{VkSubmitInfo} is in the \texttt{pNext} chain of this \texttt{VkD3D12FenceSubmitInfoKHR} structure.

- VUID-VkD3D12FenceSubmitInfoKHR-signalSemaphoreValuesCount-00080
  \textit{signalSemaphoreValuesCount} \textbf{must} be the same value as \texttt{VkSubmitInfo::signalSemaphoreCount}, where \texttt{VkSubmitInfo} is in the \texttt{pNext} chain of this \texttt{VkD3D12FenceSubmitInfoKHR} structure.

Valid Usage (Implicit)

- VUID-VkD3D12FenceSubmitInfoKHR-sType-sType
  \texttt{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHR}

- VUID-VkD3D12FenceSubmitInfoKHR-pWaitSemaphoreValues-parameter
  If \texttt{waitSemaphoreValuesCount} is not 0, and \texttt{pWaitSemaphoreValues} is not NULL, \texttt{pWaitSemaphoreValues} \textbf{must} be a valid pointer to an array of \texttt{waitSemaphoreValuesCount uint64_t values}

- VUID-VkD3D12FenceSubmitInfoKHR-pSignalSemaphoreValues-parameter
  If \texttt{signalSemaphoreValuesCount} is not 0, and \texttt{pSignalSemaphoreValues} is not NULL, \texttt{pSignalSemaphoreValues} \textbf{must} be a valid pointer to an array of \texttt{signalSemaphoreValuesCount uint64_t values}

When submitting work that operates on memory imported from a Direct3D 11 resource to a queue, the keyed mutex mechanism \textbf{may} be used in addition to Vulkan semaphores to synchronize the work. Keyed mutexes are a property of a properly created shareable Direct3D 11 resource. They \textbf{can} only be used if the imported resource was created with the \texttt{D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX} flag.

To acquire keyed mutexes before submitted work and/or release them after, add a \texttt{VkWin32KeyedMutexAcquireReleaseInfoKHR} structure to the \texttt{pNext} chain of the \texttt{VkSubmitInfo} structure.

The \texttt{VkWin32KeyedMutexAcquireReleaseInfoKHR} structure is defined as:
typedef struct VkWin32KeyedMutexAcquireReleaseInfoKHR {
    VkStructureType          sType;
    const void*              pNext;
    uint32_t                 acquireCount;
    const VkDeviceMemory*    pAcquireSyncs;
    const uint64_t*          pAcquireKeys;
    const uint32_t*          pAcquireTimeouts;
    uint32_t                 releaseCount;
    const VkDeviceMemory*    pReleaseSyncs;
    const uint64_t*          pReleaseKeys;
} VkWin32KeyedMutexAcquireReleaseInfoKHR;

• acquireCount is the number of entries in the pAcquireSyncs, pAcquireKeys, and pAcquireTimeoutMilliseconds arrays.

• pAcquireSyncs is a pointer to an array of VkDeviceMemory objects which were imported from Direct3D 11 resources.

• pAcquireKeys is a pointer to an array of mutex key values to wait for prior to beginning the submitted work. Entries refer to the keyed mutex associated with the corresponding entries in pAcquireSyncs.

• pAcquireTimeoutMilliseconds is a pointer to an array of timeout values, in millisecond units, for each acquire specified in pAcquireKeys.

• releaseCount is the number of entries in the pReleaseSyncs and pReleaseKeys arrays.

• pReleaseSyncs is a pointer to an array of VkDeviceMemory objects which were imported from Direct3D 11 resources.

• pReleaseKeys is a pointer to an array of mutex key values to set when the submitted work has completed. Entries refer to the keyed mutex associated with the corresponding entries in pReleaseSyncs.

Valid Usage

• VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-pAcquireSyncs-00081
  Each member of pAcquireSyncs and pReleaseSyncs must be a device memory object imported by setting VkImportMemoryWin32HandleInfoKHR:HANDLETYPE to VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT or VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT
Valid Usage (Implicit)

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHR`.

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-pAcquireSyncs-parameter**
  
  If `acquireCount` is not 0, `pAcquireSyncs` **must** be a valid pointer to an array of `acquireCount` valid `VkDeviceMemory` handles.

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-pAcquireKeys-parameter**
  
  If `acquireCount` is not 0, `pAcquireKeys` **must** be a valid pointer to an array of `acquireCount` `uint64_t` values.

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-pAcquireTimeouts-parameter**
  
  If `acquireCount` is not 0, `pAcquireTimeouts` **must** be a valid pointer to an array of `acquireCount` `uint32_t` values.

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-pReleaseSyncs-parameter**
  
  If `releaseCount` is not 0, `pReleaseSyncs` **must** be a valid pointer to an array of `releaseCount` valid `VkDeviceMemory` handles.

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-pReleaseKeys-parameter**
  
  If `releaseCount` is not 0, `pReleaseKeys` **must** be a valid pointer to an array of `releaseCount` `uint64_t` values.

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoKHR-commonparent**
  
  Both of the elements of `pAcquireSyncs`, and the elements of `pReleaseSyncs` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`.

When submitting work that operates on memory imported from a Direct3D 11 resource to a queue, the keyed mutex mechanism **may** be used in addition to Vulkan semaphores to synchronize the work. Keyed mutexes are a property of a properly created shareable Direct3D 11 resource. They **can** only be used if the imported resource was created with the `D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX` flag.

To acquire keyed mutexes before submitted work and/or release them after, add a `VkWin32KeyedMutexAcquireReleaseInfoNV` structure to the `pNext` chain of the `VkSubmitInfo` structure.

The `VkWin32KeyedMutexAcquireReleaseInfoNV` structure is defined as:
// Provided by VK_NV_win32_keyed_mutex
typedef struct VkWin32KeyedMutexAcquireReleaseInfoNV {
    VkStructureType sType;
    const void* pNext;
    uint32_t acquireCount;
    const VkDeviceMemory* pAcquireSyncs;
    const uint64_t* pAcquireKeys;
    const uint32_t* pAcquireTimeoutMilliseconds;
    uint32_t releaseCount;
    const VkDeviceMemory* pReleaseSyncs;
    const uint64_t* pReleaseKeys;
} VkWin32KeyedMutexAcquireReleaseInfoNV;

• acquireCount is the number of entries in the pAcquireSyncs, pAcquireKeys, and pAcquireTimeoutMilliseconds arrays.

• pAcquireSyncs is a pointer to an array of VkDeviceMemory objects which were imported from Direct3D 11 resources.

• pAcquireKeys is a pointer to an array of mutex key values to wait for prior to beginning the submitted work. Entries refer to the keyed mutex associated with the corresponding entries in pAcquireSyncs.

• pAcquireTimeoutMilliseconds is a pointer to an array of timeout values, in millisecond units, for each acquire specified in pAcquireKeys.

• releaseCount is the number of entries in the pReleaseSyncs and pReleaseKeys arrays.

• pReleaseSyncs is a pointer to an array of VkDeviceMemory objects which were imported from Direct3D 11 resources.

• pReleaseKeys is a pointer to an array of mutex key values to set when the submitted work has completed. Entries refer to the keyed mutex associated with the corresponding entries in pReleaseSyncs.
Valid Usage (Implicit)

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV`

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-pAcquireSyncs-parameter**
  - If `acquireCount` is not 0, `pAcquireSyncs` must be a valid pointer to an array of `acquireCount` valid `VkDeviceMemory` handles

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-pAcquireKeys-parameter**
  - If `acquireCount` is not 0, `pAcquireKeys` must be a valid pointer to an array of `acquireCount` `uint64_t` values

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-pAcquireTimeoutMilliseconds-parameter**
  - If `acquireCount` is not 0, `pAcquireTimeoutMilliseconds` must be a valid pointer to an array of `acquireCount` `uint32_t` values

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-pReleaseSyncs-parameter**
  - If `releaseCount` is not 0, `pReleaseSyncs` must be a valid pointer to an array of `releaseCount` valid `VkDeviceMemory` handles

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-pReleaseKeys-parameter**
  - If `releaseCount` is not 0, `pReleaseKeys` must be a valid pointer to an array of `releaseCount` `uint64_t` values

- **VUID-VkWin32KeyedMutexAcquireReleaseInfoNV-commonparent**
  - Both of the elements of `pAcquireSyncs`, and the elements of `pReleaseSyncs` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

If the `pNext` chain of `VkSubmitInfo` includes a `VkProtectedSubmitInfo` structure, then the structure indicates whether the batch is protected. The `VkProtectedSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkProtectedSubmitInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkBool32           protectedSubmit;
} VkProtectedSubmitInfo;
```

- `protectedSubmit` specifies whether the batch is protected. If `protectedSubmit` is `VK_TRUE`, the batch is protected. If `protectedSubmit` is `VK_FALSE`, the batch is unprotected. If the `VkSubmitInfo::pNext` chain does not include this structure, the batch is unprotected.

Valid Usage

- **VUID-VkProtectedSubmitInfo-protectedSubmit-01816**
  - If the protected memory feature is not enabled, `protectedSubmit` must not be `VK_TRUE`
Valid Usage (Implicit)

- VUID-VkProtectedSubmitInfo-sType-sType
  - **sType** must be `VK_STRUCTURE_TYPE_PROTECTED_SUBMIT_INFO`

If the `pNext` chain of `VkSubmitInfo` includes a `VkDeviceGroupSubmitInfo` structure, then that structure includes device indices and masks specifying which physical devices execute semaphore operations and command buffers.

The `VkDeviceGroupSubmitInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupSubmitInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           waitSemaphoreCount;
    const uint32_t*    pWaitSemaphoreDeviceIndices;
    uint32_t           commandBufferCount;
    const uint32_t*    pCommandBufferDeviceMasks;
    uint32_t           signalSemaphoreCount;
    const uint32_t*    pSignalSemaphoreDeviceIndices;
} VkDeviceGroupSubmitInfo;
```

or the equivalent

```c
// Provided by VK_KHR_device_group
typedef VkDeviceGroupSubmitInfo VkDeviceGroupSubmitInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **waitSemaphoreCount** is the number of elements in the `pWaitSemaphoreDeviceIndices` array.
- **pWaitSemaphoreDeviceIndices** is a pointer to an array of `waitSemaphoreCount` device indices indicating which physical device executes the semaphore wait operation in the corresponding element of `VkSubmitInfo::pWaitSemaphores`.
- **commandBufferCount** is the number of elements in the `pCommandBufferDeviceMasks` array.
- **pCommandBufferDeviceMasks** is a pointer to an array of `commandBufferCount` device masks indicating which physical devices execute the command buffer in the corresponding element of `VkSubmitInfo::pCommandBuffers`. A physical device executes the command buffer if the corresponding bit is set in the mask.
- **signalSemaphoreCount** is the number of elements in the `pSignalSemaphoreDeviceIndices` array.
- **pSignalSemaphoreDeviceIndices** is a pointer to an array of `signalSemaphoreCount` device indices indicating which physical device executes the semaphore signal operation in the corresponding element of `VkSubmitInfo::pSignalSemaphores`.
If this structure is not present, semaphore operations and command buffers execute on device index zero.

### Valid Usage

- **VUID-VkDeviceGroupSubmitInfo-waitSemaphoreCount-00082**
  
  \( \text{waitSemaphoreCount} \) must equal \( \text{VkSubmitInfo::waitSemaphoreCount} \)

- **VUID-VkDeviceGroupSubmitInfo-commandBufferCount-00083**
  
  \( \text{commandBufferCount} \) must equal \( \text{VkSubmitInfo::commandBufferCount} \)

- **VUID-VkDeviceGroupSubmitInfo-signalSemaphoreCount-00084**
  
  \( \text{signalSemaphoreCount} \) must equal \( \text{VkSubmitInfo::signalSemaphoreCount} \)

- **VUID-VkDeviceGroupSubmitInfo-pWaitSemaphoreDeviceIndices-00085**
  
  All elements of \( \text{pWaitSemaphoreDeviceIndices} \) and \( \text{pSignalSemaphoreDeviceIndices} \) must be valid device indices

- **VUID-VkDeviceGroupSubmitInfo-pCommandBufferDeviceMasks-00086**
  
  All elements of \( \text{pCommandBufferDeviceMasks} \) must be valid device masks

### Valid Usage (Implicit)

- **VUID-VkDeviceGroupSubmitInfo-sType-sType**
  
  \( \text{sType} \) must be \( \text{VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO} \)

- **VUID-VkDeviceGroupSubmitInfo-pWaitSemaphoreDeviceIndices-parameter**
  
  If \( \text{waitSemaphoreCount} \) is not \( 0 \), \( \text{pWaitSemaphoreDeviceIndices} \) must be a valid pointer to an array of \( \text{waitSemaphoreCount} \) uint32_t values

- **VUID-VkDeviceGroupSubmitInfo-pCommandBufferDeviceMasks-parameter**
  
  If \( \text{commandBufferCount} \) is not \( 0 \), \( \text{pCommandBufferDeviceMasks} \) must be a valid pointer to an array of \( \text{commandBufferCount} \) uint32_t values

- **VUID-VkDeviceGroupSubmitInfo-pSignalSemaphoreDeviceIndices-parameter**
  
  If \( \text{signalSemaphoreCount} \) is not \( 0 \), \( \text{pSignalSemaphoreDeviceIndices} \) must be a valid pointer to an array of \( \text{signalSemaphoreCount} \) uint32_t values

If the \( \text{pNext} \) chain of \( \text{VkSubmitInfo} \) includes a \( \text{VkPerformanceQuerySubmitInfoKHR} \) structure, then the structure indicates which counter pass is active for the batch in that submit.

The \( \text{VkPerformanceQuerySubmitInfoKHR} \) structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkPerformanceQuerySubmitInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           counterPassIndex;
} VkPerformanceQuerySubmitInfoKHR;
```
• **sType** is the type of this structure.

• **pNext** is `NULL` or a pointer to a structure extending this structure.

• **counterPassIndex** specifies which counter pass index is active.

If the `VkSubmitInfo::pNext` chain does not include this structure, the batch defaults to use counter pass index 0.

### Valid Usage

- **VUID-VkPerformanceQuerySubmitInfoKHR-counterPassIndex-03221**

  `counterPassIndex` must be less than the number of counter passes required by any queries within the batch. The required number of counter passes for a performance query is obtained by calling `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

### Valid Usage (Implicit)

- **VUID-VkPerformanceQuerySubmitInfoKHR-sType-sType**

  `sType` must be `VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR`.

### 6.6. Queue Forward Progress

When using binary semaphores, the application **must** ensure that command buffer submissions will be able to complete without any subsequent operations by the application on any queue. After any call to `vkQueueSubmit` (or other queue operation), for every queued wait on a semaphore created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` there **must** be a prior signal of that semaphore that will not be consumed by a different wait on the semaphore.

When using timeline semaphores, wait-before-signal behavior is well-defined and applications **can** submit work via `vkQueueSubmit` which defines a timeline semaphore wait operation before submitting a corresponding semaphore signal operation. For each timeline semaphore wait operation defined by a call to `vkQueueSubmit`, the application **must** ensure that a corresponding semaphore signal operation is executed before forward progress can be made.

Command buffers in the submission **can** include `vkCmdWaitEvents` commands that wait on events that will not be signaled by earlier commands in the queue. Such events **must** be signaled by the application using `vkSetEvent`, and the `vkCmdWaitEvents` commands that wait upon them **must** not be inside a render pass instance. The event **must** be set before the `vkCmdWaitEvents` command is executed.

**Note**

Implementations may have some tolerance for waiting on events to be set, but this is defined outside of the scope of Vulkan.
6.7. Secondary Command Buffer Execution

A secondary command buffer must not be directly submitted to a queue. Instead, secondary command buffers are recorded to execute as part of a primary command buffer with the command:

```c
// Provided by VK_VERSION_1_0
void vkCmdExecuteCommands(
    VkCommandBuffer commandBuffer,
    uint32_t commandBufferCount,
    const VkCommandBuffer* pCommandBuffers);
```

- `commandBuffer` is a handle to a primary command buffer that the secondary command buffers are executed in.
- `commandBufferCount` is the length of the `pCommandBuffers` array.
- `pCommandBuffers` is a pointer to an array of `commandBufferCount` secondary command buffer handles, which are recorded to execute in the primary command buffer in the order they are listed in the array.

If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, and it was recorded into any other primary command buffer which is currently in the executable or recording state, that primary command buffer becomes invalid.
Valid Usage

• VUID-vkCmdExecuteCommands-pCommandBuffers-00088
  Each element of `pCommandBuffers` must have been allocated with a level of `VK_COMMAND_BUFFER_LEVEL_SECONDARY`.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00089
  Each element of `pCommandBuffers` must be in the pending or executable state.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00091
  If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it must not be in the pending state.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00092
  If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it must not have already been recorded to `commandBuffer`.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00093
  If any element of `pCommandBuffers` was not recorded with the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag, it must not appear more than once in `pCommandBuffers`.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00094
  Each element of `pCommandBuffers` must have been allocated from a `VkCommandPool` that was created for the same queue family as the `VkCommandPool` from which `commandBuffer` was allocated.

• VUID-vkCmdExecuteCommands-contents-00095
  If `vkCmdExecuteCommands` is being called within a render pass instance, that render pass instance must have been begun with the `contents` parameter of `vkCmdBeginRenderPass` set to `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS`.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00096
  If `vkCmdExecuteCommands` is being called within a render pass instance, each element of `pCommandBuffers` must have been recorded with `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00097
  If `vkCmdExecuteCommands` is being called within a render pass instance, each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::subpass` set to the index of the subpass which the given command buffer will be executed in.

• VUID-vkCmdExecuteCommands-pInheritanceInfo-00098
  If `vkCmdExecuteCommands` is being called within a render pass instance, the render passes specified in the `pBeginInfo->pInheritanceInfo->renderPass` members of the `vkBeginCommandBuffer` commands used to begin recording each element of `pCommandBuffers` must be compatible with the current render pass.

• VUID-vkCmdExecuteCommands-pCommandBuffers-00099
  If `vkCmdExecuteCommands` is being called within a render pass instance, and any element of `pCommandBuffers` was recorded with `VkCommandBufferInheritanceInfo::framebuffer` not equal to `VK_NULL_HANDLE`, that `VkFramebuffer` must match the `VkFramebuffer` used in the current render pass instance.
If `vkCmdExecuteCommands` is being called within a render pass instance that included `VkRenderPassTransformBeginInfoQCOM` in the `pNext` chain of `VkRenderPassBeginInfo`, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceRenderPassTransformInfoQCOM` in the `pNext` chain of `VkCommandBufferBeginInfo`.

If `vkCmdExecuteCommands` is being called within a render pass instance that included `VkRenderPassTransformBeginInfoQCOM` in the `pNext` chain of `VkRenderPassBeginInfo`, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceRenderPassTransformInfoQCOM::transform` identical to `VkRenderPassTransformBeginInfoQCOM::transform`.

If `vkCmdExecuteCommands` is being called within a render pass instance that included `VkRenderPassTransformBeginInfoQCOM` in the `pNext` chain of `VkRenderPassBeginInfo`, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceRenderPassTransformInfoQCOM::renderArea` identical to `VkRenderPassBeginInfo::renderArea`.

If `vkCmdExecuteCommands` is not being called within a render pass instance, each element of `pCommandBuffers` must not have been recorded with the `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`.

If the inherited queries feature is not enabled, `commandBuffer` must not have any queries active.

If `commandBuffer` has a `VK_QUERY_TYPE_OCCLUSION` query active, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::occlusionQueryEnable` set to `VK_TRUE`.

If `commandBuffer` has a `VK_QUERY_TYPE_OCCLUSION` query active, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::queryFlags` having all bits set that are set for the query.

If `commandBuffer` has a `VK_QUERY_TYPE_PIPELINE_STATISTICS` query active, then each element of `pCommandBuffers` must have been recorded with `VkCommandBufferInheritanceInfo::pipelineStatistics` having all bits set that are set in the `VkQueryPool` the query uses.

Each element of `pCommandBuffers` must not begin any query types that are active in `commandBuffer`.

If `commandBuffer` is a protected command buffer, then each element of `pCommandBuffers` must be a protected command buffer.
If `commandBuffer` is an unprotected command buffer, then each element of `pCommandBuffers` must be an unprotected command buffer.

- VUID-vkCmdExecuteCommands-None-02286

This command must not be recorded when transform feedback is active.

## Valid Usage (Implicit)

- VUID-vkCmdExecuteCommands-commandBuffer-parameter
  commandBuffer must be a valid `VkCommandBuffer` handle

- VUID-vkCmdExecuteCommands-pCommandBuffers-parameter
  pCommandBuffers must be a valid pointer to an array of `commandBufferCount` valid `VkCommandBuffer` handles

- VUID-vkCmdExecuteCommands-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdExecuteCommands-commandBuffer-cmdpool
  The `VkCommandPool` that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdExecuteCommands-bufferlevel
  commandBuffer must be a primary `VkCommandBuffer`

- VUID-vkCmdExecuteCommands-commandBufferCount-arraylength
  commandBufferCount must be greater than 0

- VUID-vkCmdExecuteCommands-commonparent
  Both of commandBuffer, and the elements of pCommandBuffers must have been created, allocated, or retrieved from the same `VkDevice`

## Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the `VkCommandPool` that commandBuffer was allocated from must be externally synchronized

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer Graphics Compute</td>
<td></td>
</tr>
</tbody>
</table>
6.8. Command Buffer Device Mask

Each command buffer has a piece of state storing the current device mask of the command buffer. This mask controls which physical devices within the logical device all subsequent commands will execute on, including state-setting commands, action commands, and synchronization commands.

Scissor, exclusive scissor, and viewport state (excluding the count of each) can be set to different values on each physical device (only when set as dynamic state), and each physical device will render using its local copy of the state. Other state is shared between physical devices, such that all physical devices use the most recently set values for the state. However, when recording an action command that uses a piece of state, the most recent command that set that state must have included all physical devices that execute the action command in its current device mask.

The command buffer’s device mask is orthogonal to the pCommandBufferDeviceMasks member of VkDeviceGroupSubmitInfo. Commands only execute on a physical device if the device index is set in both device masks.

If the pNext chain of VkCommandBufferBeginInfo includes a VkDeviceGroupCommandBufferBeginInfo structure, then that structure includes an initial device mask for the command buffer.

The VkDeviceGroupCommandBufferBeginInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupCommandBufferBeginInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t deviceMask;
} VkDeviceGroupCommandBufferBeginInfo;
```

or the equivalent

```c
// Provided by VK_KHR_device_group
typedef VkDeviceGroupCommandBufferBeginInfo VkDeviceGroupCommandBufferBeginInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- deviceMask is the initial value of the command buffer’s device mask.

The initial device mask also acts as an upper bound on the set of devices that can ever be in the device mask in the command buffer.

If this structure is not present, the initial value of a command buffer’s device mask is set to include all physical devices in the logical device when the command buffer begins recording.
Valid Usage

- VUID-VkDeviceGroupCommandBufferBeginInfo-deviceMask-00106
  
  deviceMask must be a valid device mask value

- VUID-VkDeviceGroupCommandBufferBeginInfo-deviceMask-00107
  
  deviceMask must not be zero

Valid Usage (Implicit)

- VUID-VkDeviceGroupCommandBufferBeginInfo-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO

To update the current device mask of a command buffer, call:

```c
// Provided by VK_VERSION_1_1
void vkCmdSetDeviceMask(
    VkCommandBuffer commandBuffer,
    uint32_t deviceMask);
```

or the equivalent command

```c
// Provided by VK_KHR_device_group
void vkCmdSetDeviceMaskKHR(
    VkCommandBuffer commandBuffer,
    uint32_t deviceMask);
```

- commandBuffer is command buffer whose current device mask is modified.
- deviceMask is the new value of the current device mask.

deviceMask is used to filter out subsequent commands from executing on all physical devices whose bit indices are not set in the mask, except commands beginning a render pass instance, commands transitioning to the next subpass in the render pass instance, and commands ending a render pass instance, which always execute on the set of physical devices whose bit indices are included in the deviceMask member of the VkDeviceGroupRenderPassBeginInfo structure passed to the command beginning the corresponding render pass instance.
**Valid Usage**

- VUID-vkCmdSetDeviceMask-deviceMask-00108  
  `deviceMask` must be a valid device mask value

- VUID-vkCmdSetDeviceMask-deviceMask-00109  
  `deviceMask` must not be zero

- VUID-vkCmdSetDeviceMask-deviceMask-00110  
  `deviceMask` must not include any set bits that were not in the `VkDeviceGroupCommandBufferBeginInfo::deviceMask` value when the command buffer began recording

- VUID-vkCmdSetDeviceMask-deviceMask-00111  
  If `vkCmdSetDeviceMask` is called inside a render pass instance, `deviceMask` must not include any set bits that were not in the `VkDeviceGroupRenderPassBeginInfo::deviceMask` value when the render pass instance began recording

**Valid Usage (Implicit)**

- VUID-vkCmdSetDeviceMask-commandBuffer-parameter  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDeviceMask-commandBuffer-recording  
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetDeviceMask-commandBuffer-cmdpool  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, or transfer operations

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics Compute Transfer</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7. Synchronization and Cache Control

Synchronization of access to resources is primarily the responsibility of the application in Vulkan. The order of execution of commands with respect to the host and other commands on the device has few implicit guarantees, and needs to be explicitly specified. Memory caches and other optimizations are also explicitly managed, requiring that the flow of data through the system is largely under application control.

Whilst some implicit guarantees exist between commands, five explicit synchronization mechanisms are exposed by Vulkan:

**Fences**
- Fences can be used to communicate to the host that execution of some task on the device has completed.

**Semaphores**
- Semaphores can be used to control resource access across multiple queues.

**Events**
- Events provide a fine-grained synchronization primitive which can be signaled either within a command buffer or by the host, and can be waited upon within a command buffer or queried on the host.

**Pipeline Barriers**
- Pipeline barriers also provide synchronization control within a command buffer, but at a single point, rather than with separate signal and wait operations.

**Render Passes**
- Render passes provide a useful synchronization framework for most rendering tasks, built upon the concepts in this chapter. Many cases that would otherwise need an application to use other synchronization primitives can be expressed more efficiently as part of a render pass.

### 7.1. Execution and Memory Dependencies

An *operation* is an arbitrary amount of work to be executed on the host, a device, or an external entity such as a presentation engine. Synchronization commands introduce explicit *execution dependencies*, and *memory dependencies* between two sets of operations defined by the command’s two *synchronization scopes*.

The synchronization scopes define which other operations a synchronization command is able to create execution dependencies with. Any type of operation that is not in a synchronization command’s synchronization scopes will not be included in the resulting dependency. For example, for many synchronization commands, the synchronization scopes can be limited to just operations executing in specific *pipeline stages*, which allows other pipeline stages to be excluded from a dependency. Other scoping options are possible, depending on the particular command.
An execution dependency is a guarantee that for two sets of operations, the first set must happen-before the second set. If an operation happens-before another operation, then the first operation must complete before the second operation is initiated. More precisely:

- Let A and B be separate sets of operations.
- Let S be a synchronization command.
- Let A_S and B_S be the synchronization scopes of S.
- Let A' be the intersection of sets A and A_S.
- Let B' be the intersection of sets B and B_S.
- Submitting A, S and B for execution, in that order, will result in execution dependency E between A' and B'.
- Execution dependency E guarantees that A' happens-before B'.

An execution dependency chain is a sequence of execution dependencies that form a happens-before relation between the first dependency's A' and the final dependency's B'. For each consecutive pair of execution dependencies, a chain exists if the intersection of B_S in the first dependency and A_S in the second dependency is not an empty set. The formation of a single execution dependency from an execution dependency chain can be described by substituting the following in the description of execution dependencies:

- Let S be a set of synchronization commands that generate an execution dependency chain.
- Let A_S be the first synchronization scope of the first command in S.
- Let B_S be the second synchronization scope of the last command in S.

Execution dependencies alone are not sufficient to guarantee that values resulting from writes in one set of operations can be read from another set of operations.

Three additional types of operation are used to control memory access. Availability operations cause the values generated by specified memory write accesses to become available to a memory domain for future access. Any available value remains available until a subsequent write to the same memory location occurs (whether it is made available or not) or the memory is freed. Memory domain operations cause writes that are available to a source memory domain to become available to a destination memory domain (an example of this is making writes available to the host domain available to the device domain). Visibility operations cause values available to a memory domain to become visible to specified memory accesses.

Availability, visibility, memory domains, and memory domain operations are formally defined in the Availability and Visibility section of the Memory Model chapter. Which API operations perform each of these operations is defined in Availability, Visibility, and Domain Operations.

A memory dependency is an execution dependency which includes availability and visibility operations such that:

- The first set of operations happens-before the availability operation.
- The availability operation happens-before the visibility operation.
- The visibility operation happens-before the second set of operations.
Once written values are made visible to a particular type of memory access, they can be read or written by that type of memory access. Most synchronization commands in Vulkan define a memory dependency.

The specific memory accesses that are made available and visible are defined by the access scopes of a memory dependency. Any type of access that is in a memory dependency’s first access scope and occurs in \( A' \) is made available. Any type of access that is in a memory dependency’s second access scope and occurs in \( B' \) has any available writes made visible to it. Any type of operation that is not in a synchronization command’s access scopes will not be included in the resulting dependency.

A memory dependency enforces availability and visibility of memory accesses and execution order between two sets of operations. Adding to the description of execution dependency chains:

- Let \( a \) be the set of memory accesses performed by \( A' \).
- Let \( b \) be the set of memory accesses performed by \( B' \).
- Let \( a_s \) be the first access scope of the first command in \( S \).
- Let \( b_s \) be the second access scope of the last command in \( S \).
- Let \( a' \) be the intersection of sets \( a \) and \( a_s \).
- Let \( b' \) be the intersection of sets \( b \) and \( b_s \).
- Submitting \( A, S \) and \( B \) for execution, in that order, will result in a memory dependency \( m \) between \( A' \) and \( B' \).
- Memory dependency \( m \) guarantees that:
  - Memory writes in \( a' \) are made available.
  - Available memory writes, including those from \( a' \), are made visible to \( b' \).

Note

Execution and memory dependencies are used to solve data hazards, i.e. to ensure that read and write operations occur in a well-defined order. Write-after-read hazards can be solved with just an execution dependency, but read-after-write and write-after-write hazards need appropriate memory dependencies to be included between them. If an application does not include dependencies to solve these hazards, the results and execution orders of memory accesses are undefined.

7.1.1. Image Layout Transitions

Image subresources can be transitioned from one layout to another as part of a memory dependency (e.g. by using an image memory barrier). When a layout transition is specified in a memory dependency, it happens-after the availability operations in the memory dependency, and happens-before the visibility operations. Image layout transitions may perform read and write accesses on all memory bound to the image subresource range, so applications must ensure that all memory writes have been made available before a layout transition is executed. Available memory is automatically made visible to a layout transition, and writes performed by a layout transition are automatically made available.
Layout transitions always apply to a particular image subresource range, and specify both an old layout and new layout. The old layout must either be `VK_IMAGE_LAYOUT_UNDEFINED`, or match the current layout of the image subresource range. If the old layout matches the current layout of the image subresource range, the transition preserves the contents of that range. If the old layout is `VK_IMAGE_LAYOUT_UNDEFINED`, the contents of that range may be discarded.

As image layout transitions may perform read and write accesses on the memory bound to the image, if the image subresource affected by the layout transition is bound to peer memory for any device in the current device mask then the memory heap the bound memory comes from must support the `VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT` and `VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT` capabilities as returned by `vkGetDeviceGroupPeerMemoryFeatures`.

Note

Applications must ensure that layout transitions happen-after all operations accessing the image with the old layout, and happen-before any operations that will access the image with the new layout. Layout transitions are potentially read/write operations, so not defining appropriate memory dependencies to guarantee this will result in a data race.

Image layout transitions interact with memory aliasing.

Layout transitions that are performed via image memory barriers execute in their entirety in submission order, relative to other image layout transitions submitted to the same queue, including those performed by render passes. In effect there is an implicit execution dependency from each such layout transition to all layout transitions previously submitted to the same queue.

The image layout of each image subresource of a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` is dependent on the last sample locations used to render to the image subresource as a depth/stencil attachment, thus when the image member of an image memory barrier is an image created with this flag the application can chain a `VkSampleLocationsInfoEXT` structure to the `pNext` chain of `VkImageMemoryBarrier2KHR` or `VkImageMemoryBarrier` to specify the sample locations to use during any image layout transition.

If the `VkSampleLocationsInfoEXT` structure does not match the sample location state last used to render to the image subresource range specified by `subresourceRange`, or if no `VkSampleLocationsInfoEXT` structure is present, then the contents of the given image subresource range becomes undefined as if `oldLayout` would equal `VK_IMAGE_LAYOUT_UNDEFINED`.

### 7.1.2. Pipeline Stages

The work performed by an action or synchronization command consists of multiple operations, which are performed as a sequence of logically independent steps known as pipeline stages. The exact pipeline stages executed depend on the particular command that is used, and current command buffer state when the command was recorded. Drawing commands, dispatching commands, copy commands, clear commands, and synchronization commands all execute in different sets of pipeline stages. Synchronization commands do not execute in a defined pipeline.
Note
Operations performed by synchronization commands (e.g. availability and visibility operations) are not executed by a defined pipeline stage. However other commands can still synchronize with them by using the synchronization scopes to create a dependency chain.

Execution of operations across pipeline stages must adhere to implicit ordering guarantees, particularly including pipeline stage order. Otherwise, execution across pipeline stages may overlap or execute out of order with regards to other stages, unless otherwise enforced by an execution dependency.

Several of the synchronization commands include pipeline stage parameters, restricting the synchronization scopes for that command to just those stages. This allows fine grained control over the exact execution dependencies and accesses performed by action commands. Implementations should use these pipeline stages to avoid unnecessary stalls or cache flushing.

`VkPipelineStageFlags2KHR` is a bitmask type for setting a mask of zero or more `VkPipelineStageFlagBits2KHR` flags:

```c
// Provided by VK_KHR_synchronization2
typedef VkFlags64 VkPipelineStageFlags2KHR;
```

Flags which can be set for a `VkPipelineStageFlags2KHR` mask are:

```c
// Provided by VK_KHR_synchronization2
// Flag bits for VkPipelineStageFlagBits2KHR
typedef VkFlags64 VkPipelineStageFlags2KHR;
static const VkPipelineStageFlags2KHR VK_PIPELINE_STAGE_2_NONE_KHR = 0ULL;
static const VkPipelineStageFlags2KHR VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT_KHR = 0x00000001ULL;
static const VkPipelineStageFlags2KHR VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR = 0x00000002ULL;
static const VkPipelineStageFlags2KHR VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR = 0x00000004ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR = 0x00000008ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR = 0x00000010ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR = 0x00000020ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR = 0x00000040ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR = 0x00000080ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR = 0x00000100ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR = 0x00000200ULL;
```
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR = 0x00000400ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT_KHR = 0x00000800ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR = 0x00001000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TRANSFER_BIT_KHR = 0x00001000;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT_KHR = 0x00002000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_HOST_BIT_KHR = 0x00004000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR = 0x00008000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR = 0x00010000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COPY_BIT_KHR = 0x100000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR = 0x200000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_BLIT_BIT_KHR = 0x400000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR = 0x800000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR = 0x1000000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR = 0x2000000000ULL;
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT_KHR = 0x4000000000ULL;
#ifdef VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_decode_queue
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR = 0x04000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_encode_queue
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR = 0x08000000ULL;
#endif
// Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT = 0x010000000ULL;
// Provided by VK_KHR_synchronization2 with VK_EXT_conditional_rendering
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000ULL;
// Provided by VK_KHR_synchronization2 with VK_NV_device_generated_commands
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV = 0x00020000ULL;
// Provided by VK_KHR_synchronization2 with VK_KHR_fragment_shading_rate
static const VkPipelineStageFlagBits2KHR VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_BIT_KHR = 0x00080000ULL;

206 | Chapter 7. Synchronization and Cache Control
• **VK_PIPELINE_STAGE_2_NONE_KHR** specifies no stages of execution.

• **VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR** specifies the stage of the pipeline where indirect command parameters are consumed. This stage also includes reading commands written by `vkCmdPreprocessGeneratedCommandsNV`.

• **VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV** specifies the task shader stage.

• **VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV** specifies the mesh shader stage.

• **VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR** specifies the stage of the pipeline where index buffers are consumed.

• **VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR** specifies the stage of the pipeline where vertex buffers are consumed.

• **VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR** is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR**

• **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR** specifies the vertex shader stage.

• **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR** specifies the tessellation control shader stage.

• **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR** specifies the tessellation evaluation shader stage.
• **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR** specifies the geometry shader stage.

• **VK_PIPELINE_STAGE_2_PRE_RASTERIZATION_SHADERS_BIT_KHR** is equivalent to specifying all supported stages from:
  ◦ **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV**
  ◦ **VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV**

• **VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR** specifies the fragment shader stage.

• **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR** specifies the stage of the pipeline where early fragment tests (depth and stencil tests before fragment shading) are performed. This stage also includes subpass load operations for framebuffer attachments with a depth/stencil format.

• **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR** specifies the stage of the pipeline where late fragment tests (depth and stencil tests after fragment shading) are performed. This stage also includes subpass store operations for framebuffer attachments with a depth/stencil format.

• **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR** specifies the stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes subpass load and store operations and multisample resolve operations for framebuffer attachments with a color or depth/stencil format.

• **VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT_KHR** specifies the compute shader stage.

• **VK_PIPELINE_STAGE_2_HOST_BIT_KHR** specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.

• **VK_PIPELINE_STAGE_2_COPY_BIT_KHR** specifies the execution of all copy commands, including `vkCmdCopyQueryPoolResults`.

• **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR** specifies the execution of `vkCmdBlitImage`.

• **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR** specifies the execution of `vkCmdResolveImage`.

• **VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR** specifies the execution of clear commands, with the exception of `vkCmdClearAttachments`.

• **VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR** is equivalent to specifying all of:
  ◦ **VK_PIPELINE_STAGE_2_COPY_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR**

• **VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR** specifies the execution of the ray tracing shader stages.

• **VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR** specifies the execution of
acceleration structure commands.

- **VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR** specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:
  - **VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV**
  - **VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV**
  - **VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR**
  - **VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT**
  - **VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT**
  - **VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV**
  - **VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT**

- **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR** specifies all operations performed by all commands supported on the queue it is used with.

- **VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT** specifies the stage of the pipeline where the predicate of conditional rendering is consumed.

- **VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT** specifies the stage of the pipeline where vertex attribute output values are written to the transform feedback buffers.

- **VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV** specifies the stage of the pipeline where device-side generation of commands via `vkCmdPreprocessGeneratedCommandsNV` is handled.

- **VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** specifies the stage of the pipeline where the fragment shading rate attachment or shading rate image is read to determine the fragment shading rate for portions of a rasterized primitive.

- **VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT** specifies the stage of the pipeline where the fragment density map is read to generate the fragment areas.

- **VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR** specifies the stage of the pipeline where video decode operation are performed.

- **VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR** specifies the stage of the pipeline where video encode operation are performed.

- **VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT_KHR** is equivalent to **VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR** with `VkAccessFlags2KHR` set to 0 when specified in
the second synchronization scope, but equivalent to VK_PIPELINE_STAGE_2_NONE_KHR in the first scope.

- **VK_PIPELINE_STAGE_2_BOTTOM_OF_PIPE_BIT_KHR** is equivalent to VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR with VkAccessFlags2KHR set to 0 when specified in the first synchronization scope, but equivalent to VK_PIPELINE_STAGE_2_NONE_KHR in the second scope.

**Note**

The TOP and BOTTOM pipeline stages are deprecated, and applications should prefer VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR and VK_PIPELINE_STAGE_2_NONE_KHR.

**Note**

The VkPipelineStageFlags2KHR bitmask goes beyond the 31 individual bit flags allowable within a C99 enum, which is how VkPipelineStageFlagBits is defined. The first 31 values are common to both, and are interchangeable.

The VkPipelineStageFlagBits enum is defined as:
typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
    VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000,
    VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000,
    VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR = 0x02000000,
    VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR = 0x00200000,
    VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00400000,
    VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV = 0x00080000,
    VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV = 0x00100000,
    VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000,
    VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV = 0x00020000,
    VK_PIPELINE_STAGE_NONE_KHR = 0,
    VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV = VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR,
    VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV = VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
    VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR = VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV,
} VkPipelineStageFlagBits;
These values all have the same value/meaning as the equivalently named values for \texttt{VkPipelineStageFlags2KHR}.

- \texttt{VK_PIPELINE_STAGE_NONE_KHR} specifies no stages of execution.
- \texttt{VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT} specifies the stage of the pipeline where Draw/DispatchIndirect/TraceRaysIndirect data structures are consumed. This stage also includes reading commands written by \texttt{vkCmdExecuteGeneratedCommandsNV}.
- \texttt{VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV} specifies the task shader stage.
- \texttt{VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV} specifies the mesh shader stage.
- \texttt{VK_PIPELINE_STAGE_VERTEX_INPUT_BIT} specifies the stage of the pipeline where vertex and index buffers are consumed.
- \texttt{VK_PIPELINE_STAGE_VERTEX_SHADER_BIT} specifies the vertex shader stage.
- \texttt{VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT} specifies the tessellation control shader stage.
- \texttt{VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT} specifies the tessellation evaluation shader stage.
- \texttt{VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT} specifies the geometry shader stage.
- \texttt{VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT} specifies the fragment shader stage.
- \texttt{VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT} specifies the stage of the pipeline where early fragment tests (depth and stencil tests before fragment shading) are performed. This stage also includes subpass load operations for framebuffer attachments with a depth/stencil format.
- \texttt{VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT} specifies the stage of the pipeline where late fragment tests (depth and stencil tests after fragment shading) are performed. This stage also includes subpass store operations for framebuffer attachments with a depth/stencil format.
- \texttt{VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT} specifies the stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes subpass load and store operations and multisample resolve operations for framebuffer attachments with a color or depth/stencil format.
- \texttt{VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT} specifies the execution of a compute shader.
- \texttt{VK_PIPELINE_STAGE_TRANSFER_BIT} specifies the following commands:
  - All copy commands, including \texttt{vkCmdCopyQueryPoolResults}
  - \texttt{vkCmdBlitImage2KHR} and \texttt{vkCmdBlitImage}
  - \texttt{vkCmdResolveImage2KHR} and \texttt{vkCmdResolveImage}
  - All clear commands, with the exception of \texttt{vkCmdClearAttachments}
- \texttt{VK_PIPELINE_STAGE_HOST_BIT} specifies a pseudo-stage indicating execution on the host of reads/writes of device memory. This stage is not invoked by any commands recorded in a command buffer.
- \texttt{VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR} specifies the execution of \texttt{vkCmdBuildAccelerationStructureNV}, \texttt{vkCmdCopyAccelerationStructureNV}, \texttt{vkCmdWriteAccelerationStructuresPropertiesNV}, \texttt{vkCmdBuildAccelerationStructuresKHR},...
• **VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR** specifies the execution of the ray tracing shader stages, via `vkCmdTraceRaysNV`, `vkCmdTraceRaysKHR`, or `vkCmdTraceRaysIndirectKHR`.

• **VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT** specifies the execution of all graphics pipeline stages, and is equivalent to the logical OR of:

  ◦ **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT**
  ◦ **VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV**
  ◦ **VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV**
  ◦ **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT**
  ◦ **VK_PIPELINE_STAGE_VERTEX_SHADER_BIT**
  ◦ **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT**
  ◦ **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT**
  ◦ **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**
  ◦ **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT**
  ◦ **VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT**
  ◦ **VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT**
  ◦ **VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT**
  ◦ **VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT**
  ◦ **VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT**
  ◦ **VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**
  ◦ **VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT**

• **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** specifies all operations performed by all commands supported on the queue it is used with.

• **VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT** specifies the stage of the pipeline where the predicate of conditional rendering is consumed.

• **VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT** specifies the stage of the pipeline where vertex attribute output values are written to the transform feedback buffers.

• **VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV** specifies the stage of the pipeline where device-side preprocessing for generated commands via `vkCmdPreprocessGeneratedCommandsNV` is handled.

• **VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR** specifies the stage of the pipeline where the fragment shading rate attachment or shading rate image is read to determine the fragment shading rate for portions of a rasterized primitive.

• **VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT** specifies the stage of the pipeline where the fragment density map is read to generate the fragment areas.
- **VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT** is equivalent to **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** with **VkAccessFlags** set to 0 when specified in the second synchronization scope, but specifies no stage of execution when specified in the first scope.

- **VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT** is equivalent to **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** with **VkAccessFlags** set to 0 when specified in the first synchronization scope, but specifies no stage of execution when specified in the second scope.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineStageFlags;
```

**VkPipelineStageFlags** is a bitmask type for setting a mask of zero or more **VkPipelineStageFlagBits**.

If a synchronization command includes a source stage mask, its first **synchronization scope** only includes execution of the pipeline stages specified in that mask, and its first **access scope** only includes memory access performed by pipeline stages specified in that mask. If a synchronization command includes a destination stage mask, its second **synchronization scope** only includes execution of the pipeline stages specified in that mask, and its second **access scope** only includes memory access performed by pipeline stages specified in that mask.

**Note**

Including a particular pipeline stage in the first **synchronization scope** of a command implicitly includes **logically earlier** pipeline stages in the synchronization scope. Similarly, the second **synchronization scope** includes **logically later** pipeline stages.

However, note that **access scopes** are not affected in this way - only the precise stages specified are considered part of each access scope.

Certain pipeline stages are only available on queues that support a particular set of operations. The following table lists, for each pipeline stage flag, which queue capability flag **must** be supported by the queue. When multiple flags are enumerated in the second column of the table, it means that the pipeline stage is supported on the queue if it supports any of the listed capability flags. For further details on queue capabilities see **Physical Device Enumeration** and **Queues**.

**Table 3. Supported pipeline stage flags**

<table>
<thead>
<tr>
<th>Pipeline stage flag</th>
<th>Required queue capability flag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</strong></td>
<td>None required</td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</strong></td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT</strong></td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT</strong></td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT</strong></td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT</strong></td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT</strong></td>
</tr>
<tr>
<td><strong>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</strong></td>
<td><strong>VK_QUEUE_GRAPHICS_BIT</strong></td>
</tr>
<tr>
<td>Pipeline stage flag</td>
<td>Required queue capability flag</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT, VK_QUEUE_COMPUTE_BIT or VK_QUEUE_TRANSFER_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ALL_COMMANDS_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR</td>
<td>VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT_KHR</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
</tbody>
</table>

Pipeline stages that execute as a result of a command logically complete execution in a specific order, such that completion of a logically later pipeline stage must not happen-before completion of a logically earlier stage. This means that including any stage in the source stage mask for a particular synchronization command also implies that any logically earlier stages are included in $A_s$ for that command.

Similarly, initiation of a logically earlier pipeline stage must not happen-after initiation of a logically later pipeline stage. Including any given stage in the destination stage mask for a particular synchronization command also implies that any logically later stages are included in $B_s$ for that command.
Note

Implementations may not support synchronization at every pipeline stage for every synchronization operation. If a pipeline stage that an implementation does not support synchronization for appears in a source stage mask, it may substitute any logically later stage in its place for the first synchronization scope. If a pipeline stage that an implementation does not support synchronization for appears in a destination stage mask, it may substitute any logically earlier stage in its place for the second synchronization scope.

For example, if an implementation is unable to signal an event immediately after vertex shader execution is complete, it may instead signal the event after color attachment output has completed.

If an implementation makes such a substitution, it must not affect the semantics of execution or memory dependencies or image and buffer memory barriers.

Graphics pipelines are executable on queues supporting VK_QUEUE_GRAPHICS_BIT. Stages executed by graphics pipelines can only be specified in commands recorded for queues supporting VK_QUEUE_GRAPHICS_BIT.

The graphics primitive pipeline executes the following stages, with the logical ordering of the stages matching the order specified here:

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR
- VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT
- VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

The graphics mesh pipeline executes the following stages, with the logical ordering of the stages matching the order specified here:

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV
- VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV
• VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR
• VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
• VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
• VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
• VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

For the compute pipeline, the following stages occur in this order:

• VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
• VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT

For graphics pipeline commands executing in a render pass with a fragment density map attachment, the following pipeline stage where the fragment density map read happens has no particular order relative to the other stages, except that it is logically earlier than VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT:

• VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT
• VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT

The conditional rendering stage is formally part of both the graphics, and the compute pipeline. The pipeline stage where the predicate read happens has unspecified order relative to other stages of these pipelines:

• VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT

For the transfer pipeline, the following stages occur in this order:

• VK_PIPELINE_STAGE_TRANSFER_BIT

For host operations, only one pipeline stage occurs, so no order is guaranteed:

• VK_PIPELINE_STAGE_HOST_BIT

For the command preprocessing pipeline, the following stages occur in this order:

• VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV

For acceleration structure operations, only one pipeline stage occurs, so no order is guaranteed:

• VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR

For the ray tracing pipeline, the following stages occur in this order:

• VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
• VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR

7.1.3. Access Types

Memory in Vulkan can be accessed from within shader invocations and via some fixed-function
stages of the pipeline. The *access type* is a function of the *descriptor type* used, or how a fixed-function stage accesses memory.

Some synchronization commands take sets of access types as parameters to define the *access scopes* of a memory dependency. If a synchronization command includes a source access mask, its first *access scope* only includes accesses via the access types specified in that mask. Similarly, if a synchronization command includes a destination access mask, its second *access scope* only includes accesses via the access types specified in that mask.

`VkAccessFlags2KHR` is a bitmask type for setting a mask of zero or more `VkAccessFlagBits2KHR`:

```c
// Provided by VK_KHR_synchronization2
typedef VkFlags64 VkAccessFlags2KHR;
```

Flags which can be set for a `VkAccessFlags2KHR` mask are:

```c
// Provided by VK_KHR_synchronization2
// Flag bits for VkAccessFlagBits2KHR
typedef VkFlags64 VkAccessFlagBits2KHR;
static const VkAccessFlagBits2KHR VK_ACCESS_2_NONE_KHR = 0ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR = 0x00000001ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_INDEX_READ_BIT_KHR = 0x00000002ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR = 0x00000004ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_UNIFORM_READ_BIT_KHR = 0x00000008ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR = 0x00000010ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_READ_BIT_KHR = 0x00000020ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_WRITE_BIT_KHR = 0x00000040ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR = 0x00000080ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR = 0x00000100ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR = 0x00000200ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR = 0x00000400ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFER_READ_BIT_KHR = 0x00000800ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR = 0x00001000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_HOST_READ_BIT_KHR = 0x00002000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_HOST_WRITE_BIT_KHR = 0x00004000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_MEMORY_READ_BIT_KHR = 0x00008000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_MEMORY_WRITE_BIT_KHR = 0x00010000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR = 0x100000000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR = 0x200000000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR = 0x400000000ULL;
```
```c
0x400000000ULL;
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_decode_queue
    static const VkAccessFlagBits2KHR VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR = 0x800000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_decode_queue
    static const VkAccessFlagBits2KHR VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR = 0x1000000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_encode_queue
    static const VkAccessFlagBits2KHR VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR = 0x2000000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_encode_queue
    static const VkAccessFlagBits2KHR VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR = 0x4000000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
    static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT = 0x02000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
    static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT = 0x04000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_synchronization2 with VK_EXT_transform_feedback
    static const VkAccessFlagBits2KHR VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT = 0x08000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_synchronization2 with VK_KHR_fragment_shading_rate
    static const VkAccessFlagBits2KHR VK_ACCESS_2_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR = 0x008000000ULL;
#endif
#ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_synchronization2 with VK_KHR_acceleration_structure
    static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR = 0x00200000ULL;
#endif
```
static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_NV = 0x00200000;
static const VkAccessFlagBits2KHR VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_NV = 0x00400000;
static const VkAccessFlagBits2KHR VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT = 0x01000000ULL;
static const VkAccessFlagBits2KHR VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT = 0x00080000ULL;

• VK_ACCESS_2_NONE_KHR specifies no accesses.
• VK_ACCESS_2_MEMORY_READ_BIT_KHR specifies all read accesses. It is always valid in any access mask, and is treated as equivalent to setting all READ access flags that are valid where it is used.
• VK_ACCESS_2_MEMORY_WRITE_BIT_KHR specifies all write accesses. It is always valid in any access mask, and is treated as equivalent to setting all WRITE access flags that are valid where it is used.
• VK_ACCESS_2 INDIRECT_COMMAND_READ_BIT_KHR specifies read access to indirect buffers in the VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR pipeline stage.
• VK_ACCESS_2_INDEX_READ_BIT_KHR specifies read access to an index buffer in the VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR pipeline stage.
• VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR specifies read access to a vertex buffer in the VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR pipeline stage.
• VK_ACCESS_2_UNIFORM_READ_BIT_KHR specifies read access to a uniform buffer in any shader pipeline stage.
• VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR specifies read access to an input attachment in the VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR pipeline stage.
• VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR specifies read access to a uniform texel buffer or sampled image in any shader pipeline stage.
• VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR specifies read access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.
• VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR specifies write access to a storage buffer, physical storage buffer, storage texel buffer, or storage image in any shader pipeline stage.
• VK_ACCESS_2_SHADER_READ_BIT_KHR is equivalent to the logical OR of:
  ◦ VK_ACCESS_2_UNIFORM_READ_BIT_KHR
  ◦ VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR
  ◦ VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR
• VK_ACCESS_2_SHADER_WRITE_BIT_KHR is equivalent to VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR.
• VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR specifies read access to a color attachment (excluding advanced blend operations) in the VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR pipeline stage.
• **VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR** specifies write access to a color or resolve attachment in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR** specifies read access to a depth/stencil attachment in the **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR** or **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR** specifies write access to a depth/stencil attachment in the **VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR** or **VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_TRANSFER_READ_BIT_KHR** specifies read access to an image or buffer in the **VK_PIPELINE_STAGE_2_COPY_BIT_KHR**, **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR**, or **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR** specifies write access to an image or buffer in the **VK_PIPELINE_STAGE_2_COPY_BIT_KHR**, **VK_PIPELINE_STAGE_2_BLIT_BIT_KHR**, **VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR**, or **VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_HOST_READ_BIT_KHR** specifies read access to memory in the **VK_PIPELINE_STAGE_2_HOST_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_HOST_WRITE_BIT_KHR** specifies write access to memory in the **VK_PIPELINE_STAGE_2_HOST_BIT_KHR** pipeline stage.

• **VK_ACCESS_2CONDITIONAL_RENDERING_READ_BIT_EXT** specifies read access to a predicate in the **VK_PIPELINE_STAGE_2CONDITIONAL_RENDERING_BIT_EXT** pipeline stage.

• **VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT** specifies write access to a transform feedback buffer in the **VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT** pipeline stage.

• **VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT** specifies read access to a transform feedback counter buffer in the **VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT** pipeline stage.

• **VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT** specifies write access to a transform feedback counter buffer in the **VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT** pipeline stage.

• **VK_ACCESS_2_COMMANDPREPROCESS_READ_BIT_NV** specifies reads from VkBuffer inputs to **vkCmdPreprocessGeneratedCommandsNV** in the **VK_PIPELINE_STAGE_2_COMMANDPREPROCESS_BIT_NV** pipeline stage.

• **VK_ACCESS_2_COMMANDPREPROCESS_WRITE_BIT_NV** specifies writes to the target command buffer in the **VK_PIPELINE_STAGE_2_COMMANDPREPROCESS_BIT_NV** pipeline stage.

• **VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT** specifies read access to color attachments, including advanced blend operations, in the **VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR** pipeline stage.

• **VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV** specifies read access to a shading rate image in the **VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV** pipeline stage.

• **VK_ACCESS_2_ACCELERATIONSTRUCTURE_READ_BIT_KHR** specifies read access to an acceleration structure in the **VK_PIPELINE_STAGE_2_ACCELERATIONSTRUCTUREBUILD_BIT_KHR** or **VK_PIPELINE_STAGE_2_RAYTRACING_SHADER_BIT_KHR** pipeline stages.

• **VK_ACCESS_2_ACCELERATIONSTRUCTURE_WRITE_BIT_KHR** specifies write access to an acceleration structure in the **VK_PIPELINE_STAGE_2_ACCELERATIONSTRUCTUREBUILD_BIT_KHR** pipeline stage.
• **VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT** specifies read access to a fragment density map attachment in the **VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT** pipeline stage.

• **VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR** specifies read access to an image or buffer resource as part of a video decode operation.

• **VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR** specifies write access to an image or buffer resource as part of a video decode operation.

• **VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR** specifies read access to an image or buffer resource as part of a video encode operation.

• **VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR** specifies write access to an image or buffer resource as part of a video encode operation.

**Note**

In situations where an application wishes to select all access types for a given set of pipeline stages, **VK_ACCESS_2_MEMORY_READ_BIT_KHR** or **VK_ACCESS_2_MEMORY_WRITE_BIT_KHR** can be used. This is particularly useful when specifying stages that only have a single access type.

**Note**

The **VkAccessFlags2KHR** bitmask goes beyond the 31 individual bit flags allowable within a C99 enum, which is how **VkAccessFlagBits** is defined. The first 31 values are common to both, and are interchangeable.

The **VkAccessFlagBits** enums is defined as:

```c
// Provided by VK_VERSION_1_0
typedef enum VkAccessFlagBits {
    VK_ACCESS_INDIRECT_COMMAND_READ_BIT = 0x00000001,
    VK_ACCESS_INDEX_READ_BIT = 0x00000002,
    VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT = 0x00000004,
    VK_ACCESS_UNIFORM_READ_BIT = 0x00000008,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT = 0x00000010,
    VK_ACCESS_SHADER_READ_BIT = 0x00000020,
    VK_ACCESS_SHADER_WRITE_BIT = 0x00000040,
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT = 0x00000080,
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT = 0x00000100,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT = 0x00000200,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT = 0x00000400,
    VK_ACCESS_TRANSFER_READ_BIT = 0x00000800,
    VK_ACCESS_TRANSFER_WRITE_BIT = 0x00001000,
    VK_ACCESS_HOST_READ_BIT = 0x00002000,
    VK_ACCESS_HOST_WRITE_BIT = 0x00004000,
    VK_ACCESS_MEMORY_READ_BIT = 0x00008000,
    VK_ACCESS_MEMORY_WRITE_BIT = 0x00010000,
    // Provided by VK_EXT_transform_feedback
    VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT = 0x02000000,
    // Provided by VK_EXT_transform_feedback
```
VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT = 0x04000000,
// Provided by VK_EXT_transform_feedback
VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT = 0x08000000,
// Provided by VK_EXT_transform_feedback
VK_ACCESS_CONDITIONAL_RENDERING_READ_BIT_EXT = 0x00100000,
// Provided by VK_EXT_conditional_rendering
VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT = 0x00080000,
// Provided by VK_KHR_acceleration_structure
VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR = 0x00200000,
// Provided by VK_KHR_acceleration_structure
VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR = 0x00400000,
// Provided by VK_KHR_acceleration_structure
VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV = 0x00800000,
// Provided by VK_EXT_fragment_density_map
VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT = 0x01000000,
// Provided by VK_NV_device_generated_commands
VK_ACCESS_COMMAND_PREPROCESS_READ_BIT_NV = 0x00020000,
// Provided by VK_NV_device_generated_commands
VK_ACCESS_COMMAND_PREPROCESS_WRITE_BIT_NV = 0x00040000,
// Provided by VK_KHR_synchronization2
VK_ACCESS_NONE_KHR = 0,
// Provided by VK_NV_ray_tracing
VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV = VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR,
// Provided by VK_KHR_acceleration_structure
VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV = VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR,
// Provided by VK_KHR_acceleration_structure
VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT = VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV,
// Provided by VK_EXT_device_memory
VK_ACCESS_SHADER_READ_BIT
// Specifies read access to a storage buffer, physical storage buffer, shader binding table, uniform texel buffer, storage texel buffer, sampled image, or storage

These values all have the same value/meaning as the equivalently named values for VkAccessFlags2KHR.

- **VK_ACCESS_NONE_KHR** specifies no accesses.
- **VK_ACCESS_INDIRECT_COMMAND_READ_BIT** specifies read access to indirect command data read as part of an indirect build, trace, drawing or dispatch command.
- **VK_ACCESS_INDEX_READ_BIT** specifies read access to an index buffer as part of an indexed drawing command, bound by vkCmdBindIndexBuffer.
- **VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT** specifies read access to a vertex buffer as part of a drawing command, bound by vkCmdBindVertexBuffer.
- **VK_ACCESS_UNIFORM_READ_BIT** specifies read access to a uniform buffer.
- **VK_ACCESS_INPUT_ATTACHMENT_READ_BIT** specifies read access to an input attachment within a render pass during fragment shading.
- **VK_ACCESS_SHADER_READ_BIT** specifies read access to a storage buffer, physical storage buffer, shader binding table, uniform texel buffer, storage texel buffer, sampled image, or storage...
VK_ACCESS_SHADER_WRITE_BIT specifies write access to a storage buffer, physical storage buffer, storage texel buffer, or storage image.

VK_ACCESS_COLOR_ATTACHMENT_READ_BIT specifies read access to a color attachment, such as via blending, logic operations, or via certain subpass load operations. It does not include advanced blend operations.

VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT specifies write access to a color, resolve, or depth/stencil resolve attachment during a render pass or via certain subpass load and store operations.

VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT specifies read access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load operations.

VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT specifies write access to a depth/stencil attachment, via depth or stencil operations or via certain subpass load and store operations.

VK_ACCESS_TRANSFER_READ_BIT specifies read access to an image or buffer in a copy operation.

VK_ACCESS_TRANSFER_WRITE_BIT specifies write access to an image or buffer in a clear or copy operation.

VK_ACCESS_HOST_READ_BIT specifies read access by a host operation. Accesses of this type are not performed through a resource, but directly on memory.

VK_ACCESS_HOST_WRITE_BIT specifies write access by a host operation. Accesses of this type are not performed through a resource, but directly on memory.

VK_ACCESS_MEMORY_READ_BIT specifies all read accesses. It is always valid in any access mask, and is treated as equivalent to setting all READ access flags that are valid where it is used.

VK_ACCESS_MEMORY_WRITE_BIT specifies all write accesses. It is always valid in any access mask, and is treated as equivalent to setting all WRITE access flags that are valid where it is used.

VK_ACCESS_CONDITIONAL_RENDERING_READ_BIT_EXT specifies read access to a predicate as part of conditional rendering.

VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT specifies write access to a transform feedback buffer made when transform feedback is active.

VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT specifies read access to a transform feedback counter buffer which is read when vkCmdBeginTransformFeedbackEXT executes.

VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT specifies write access to a transform feedback counter buffer which is written when vkCmdEndTransformFeedbackEXT executes.

VK_ACCESS_COMMAND_PREPROCESS_READ_BIT_NV specifies reads from VkBuffer inputs to vkCmdPreprocessGeneratedCommandsNV.

VK_ACCESS_COMMAND_PREPROCESS_WRITE_BIT_NV specifies writes to the VkBuffer preprocess outputs in vkCmdPreprocessGeneratedCommandsNV.

VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT is similar to VK_ACCESS_COLOR_ATTACHMENT_READ_BIT, but also includes advanced blend operations.

VK_ACCESS_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR specifies read access to a fragment shading rate attachment or shading rate image during rasterization.

VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR specifies read access to an acceleration image.
structure as part of a trace, build, or copy command, or to an acceleration structure scratch buffer as part of a build command.

- **VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR** specifies write access to an acceleration structure or acceleration structure scratch buffer as part of a build or copy command.

- **VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT** specifies read access to a fragment density map attachment during dynamic fragment density map operations

Certain access types are only performed by a subset of pipeline stages. Any synchronization command that takes both stage masks and access masks uses both to define the access scopes - only the specified access types performed by the specified stages are included in the access scope. An application must not specify an access flag in a synchronization command if it does not include a pipeline stage in the corresponding stage mask that is able to perform accesses of that type. The following table lists, for each access flag, which pipeline stages can perform that type of access.

### Table 4. Supported access types

<table>
<thead>
<tr>
<th>Access flag</th>
<th>Supported pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_INDIRECT_COMMAND_READ_BIT</td>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT, VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR</td>
</tr>
<tr>
<td>VK_ACCESS_INDEX_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT</td>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV, VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV, VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV, VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV, VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>Access flag</td>
<td>Supported pipeline stages</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VK_ACCESS_SHADER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV, VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV, VK_PIPELINE_STAGERAY_TRACING_SHADER_BIT_KHR, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGETESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, or VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_INPUT_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT, or VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_READ_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT or VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT or VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_READ_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_HOST_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_HOST_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_READ_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_MEMORY_WRITE_BIT</td>
<td>Any</td>
</tr>
<tr>
<td>VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT</td>
<td>VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT</td>
</tr>
<tr>
<td>VK_ACCESS_COMMAND_PREPROCESS_READ_BIT_NV</td>
<td>VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV</td>
</tr>
<tr>
<td>VK_ACCESS_COMMAND_PREPROCESS_WRITE_BIT_NV</td>
<td>VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV</td>
</tr>
<tr>
<td>VK_ACCESSCONDITIONAL_RENDERING_READ_BIT_EXT</td>
<td>VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT</td>
</tr>
<tr>
<td>VK_ACCESS_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT</td>
<td>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT_EXT</td>
<td>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT_EXT</td>
<td>VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT, VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
</tr>
<tr>
<td>Access flag</td>
<td>Supported pipeline stages</td>
</tr>
<tr>
<td>----------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR</td>
<td>VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,</td>
</tr>
<tr>
<td></td>
<td>VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR,</td>
</tr>
<tr>
<td>or</td>
<td>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR</td>
</tr>
<tr>
<td>VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR</td>
<td>VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR</td>
</tr>
<tr>
<td>VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT</td>
<td>VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT</td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_0
typedef VkFlags VkAccessFlags;

VkAccessFlags is a bitmask type for setting a mask of zero or more VkAccessFlagBits.

If a memory object does not have the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT property, then vkFlushMappedMemoryRanges must be called in order to guarantee that writes to the memory object from the host are made available to the host domain, where they can be further made available to the device domain via a domain operation. Similarly, vkInvalidateMappedMemoryRanges must be called to guarantee that writes which are available to the host domain are made visible to host operations.

If the memory object does have the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT property flag, writes to the memory object from the host are automatically made available to the host domain. Similarly, writes made available to the host domain are automatically made visible to the host.

Note
Queue submission commands automatically perform a domain operation from host to device for all writes performed before the command executes, so in most cases an explicit memory barrier is not needed for this case. In the few circumstances where a submit does not occur between the host write and the device read access, writes can be made available by using an explicit memory barrier.

7.1.4. Framebuffer Region Dependencies

Pipeline stages that operate on, or with respect to, the framebuffer are collectively the framebuffer-space pipeline stages. These stages are:
• VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
• VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
• VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
• VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

For these pipeline stages, an execution or memory dependency from the first set of operations to the second set can either be a single framebuffer-global dependency, or split into multiple framebuffer-local dependencies. A dependency with non-framebuffer-space pipeline stages is neither framebuffer-global nor framebuffer-local.

A framebuffer region is a subset of the entire framebuffer, and can either be:

• A sample region, which is set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer, or
• A fragment region, which is a set of fragment (x, y, layer) coordinates that is a subset of the entire framebuffer.

Both synchronization scopes of a framebuffer-local dependency include only the operations performed within corresponding framebuffer regions (as defined below). No ordering guarantees are made between different framebuffer regions for a framebuffer-local dependency.

Both synchronization scopes of a framebuffer-global dependency include operations on all framebuffer-regions.

If the first synchronization scope includes operations on pixels/fragments with N samples and the second synchronization scope includes operations on pixels/fragments with M samples, where N does not equal M, then a framebuffer region containing all samples at a given (x, y, layer) coordinate in the first synchronization scope corresponds to a region containing all samples at the same coordinate in the second synchronization scope. In other words, the framebuffer region is a fragment region and it is a pixel granularity dependency. If N equals M, and if the VkSubpassDescription::flags does not specify the VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM flag, then a framebuffer region containing a single (x, y, layer, sample) coordinate in the first synchronization scope corresponds to a region containing the same sample at the same coordinate in the second synchronization scope. In other words, the framebuffer region is a sample region and it is a sample granularity dependency.

Note

Since fragment invocations are not specified to run in any particular groupings, the size of a framebuffer region is implementation-dependent, not known to the application, and must be assumed to be no larger than specified above.
Practically, the pixel vs sample granularity dependency means that if an input attachment has a different number of samples than the pipeline's `rasterizationSamples`, then a fragment can access any sample in the input attachment's pixel even if it only uses framebuffer-local dependencies. If the input attachment has the same number of samples, then the fragment can only access the covered samples in its input `SampleMask` (i.e. the fragment operations happen-after a framebuffer-local dependency for each sample the fragment covers). To access samples that are not covered, either the `VkSubpassDescription::flags VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM` flag is required, or a framebuffer-global dependency is required.

If a synchronization command includes a `dependencyFlags` parameter, and specifies the `VK_DEPENDENCY_BY_REGION_BIT` flag, then it defines framebuffer-local dependencies for the framebuffer-space pipeline stages in that synchronization command, for all framebuffer regions. If no `dependencyFlags` parameter is included, or the `VK_DEPENDENCY_BY_REGION_BIT` flag is not specified, then a framebuffer-global dependency is specified for those stages. The `VK_DEPENDENCY_BY_REGION_BIT` flag does not affect the dependencies between non-framebuffer-space pipeline stages, nor does it affect the dependencies between framebuffer-space and non-framebuffer-space pipeline stages.

Framebuffer-local dependencies are more optimal for most architectures; particularly tile-based architectures - which can keep framebuffer-regions entirely in on-chip registers and thus avoid external bandwidth across such a dependency. Including a framebuffer-global dependency in your rendering will usually force all implementations to flush data to memory, or to a higher level cache, breaking any potential locality optimizations.

### 7.1.5. View-Local Dependencies

In a render pass instance that has `multiview` enabled, dependencies can be either view-local or view-global.

A view-local dependency only includes operations from a single source view from the source subpass in the first synchronization scope, and only includes operations from a single destination view from the destination subpass in the second synchronization scope. A view-global dependency includes all views in the view mask of the source and destination subpasses in the corresponding synchronization scopes.

If a synchronization command includes a `dependencyFlags` parameter and specifies the `VK_DEPENDENCY_VIEW_LOCAL_BIT` flag, then it defines view-local dependencies for that synchronization command, for all views. If no `dependencyFlags` parameter is included or the `VK_DEPENDENCY_VIEW_LOCAL_BIT` flag is not specified, then a view-global dependency is specified.
7.1.6. Device-Local Dependencies

Dependencies can be either device-local or non-device-local. A device-local dependency acts as multiple separate dependencies, one for each physical device that executes the synchronization command, where each dependency only includes operations from that physical device in both synchronization scopes. A non-device-local dependency is a single dependency where both synchronization scopes include operations from all physical devices that participate in the synchronization command. For subpass dependencies, all physical devices in the `VkDeviceGroupRenderPassBeginInfo::deviceMask` participate in the dependency, and for pipeline barriers all physical devices that are set in the command buffer's current device mask participate in the dependency.

If a synchronization command includes a `dependencyFlags` parameter and specifies the `VK_DEPENDENCY_DEVICE_GROUP_BIT` flag, then it defines a non-device-local dependency for that synchronization command. If no `dependencyFlags` parameter is included or the `VK_DEPENDENCY_DEVICE_GROUP_BIT` flag is not specified, then it defines device-local dependencies for that synchronization command, for all participating physical devices.

Semaphore and event dependencies are device-local and only execute on the one physical device that performs the dependency.

7.2. Implicit Synchronization Guarantees

A small number of implicit ordering guarantees are provided by Vulkan, ensuring that the order in which commands are submitted is meaningful, and avoiding unnecessary complexity in common operations.

Submission order is a fundamental ordering in Vulkan, giving meaning to the order in which action and synchronization commands are recorded and submitted to a single queue. Explicit and implicit ordering guarantees between commands in Vulkan all work on the premise that this ordering is meaningful. This order does not itself define any execution or memory dependencies; synchronization commands and other orderings within the API use this ordering to define their scopes.

Submission order for any given set of commands is based on the order in which they were recorded to command buffers and then submitted. This order is determined as follows:

1. The initial order is determined by the order in which `vkQueueSubmit` and `vkQueueSubmit2KHR` commands are executed on the host, for a single queue, from first to last.
2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, or in which `VkSubmitInfo2KHR` structures are specified in the `pSubmits` parameter of `vkQueueSubmit2KHR`, from lowest index to highest.
3. The order in which command buffers are specified in the `pCommandBuffers` member of `VkSubmitInfo` or `VkSubmitInfo2KHR` from lowest index to highest.
4. The order in which commands were recorded to a command buffer on the host, from first to last:
   - For commands recorded outside a render pass, this includes all other commands recorded
outside a render pass, including `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands; it does not directly include commands inside a render pass.

- For commands recorded inside a render pass, this includes all other commands recorded inside the same subpass, including the `vkCmdBeginRenderPass` and `vkCmdEndRenderPass` commands that delimit the same render pass instance; it does not include commands recorded to other subpasses. **State commands** do not execute any operations on the device, instead they set the state of the command buffer when they execute on the host, in the order that they are recorded. **Action commands** consume the current state of the command buffer when they are recorded, and will execute state changes on the device as required to match the recorded state.

Query commands, the order of primitives passing through the graphics pipeline and image layout transitions as part of an image memory barrier provide additional guarantees based on submission order.

Execution of **pipeline stages** within a given command also has a loose ordering, dependent only on a single command.

**Signal operation order** is a fundamental ordering in Vulkan, giving meaning to the order in which semaphore and fence signal operations occur when submitted to a single queue. The signal operation order for queue operations is determined as follows:

1. The initial order is determined by the order in which `vkQueueSubmit` commands are executed on the host, for a single queue, from first to last.
2. The order in which `VkSubmitInfo` structures are specified in the `pSubmits` parameter of `vkQueueSubmit`, from lowest index to highest.
3. The fence signal operation defined by the `fence` parameter of a `vkQueueSubmit` or `vkQueueBindSparse` command is ordered after all semaphore signal operations defined by that command.

Semaphore signal operations defined by a single `VkSubmitInfo` or `VkBindSparseInfo` structure are unordered with respect to other semaphore signal operations defined within the same `VkSubmitInfo` or `VkBindSparseInfo` structure.

The `vkSignalSemaphore` command does not execute on a queue but instead performs the signal operation from the host. The semaphore signal operation defined by executing a `vkSignalSemaphore` command happens-after the `vkSignalSemaphore` command is invoked and happens-before the command returns.
When signaling timeline semaphores, it is the responsibility of the application to ensure that they are ordered such that the semaphore value is strictly increasing. Because the first synchronization scope for a semaphore signal operation contains all semaphore signal operations which occur earlier in submission order, all semaphore signal operations contained in any given batch are guaranteed to happen-after all semaphore signal operations contained in any previous batches. However, no ordering guarantee is provided between the semaphore signal operations defined within a single batch. This, combined with the requirement that timeline semaphore values strictly increase, means that it is invalid to signal the same timeline semaphore twice within a single batch.

If an application wishes to ensure that some semaphore signal operation happens-after some other semaphore signal operation, it can submit a separate batch containing only semaphore signal operations, which will happen-after the semaphore signal operations in any earlier batches.

When signaling a semaphore from the host, the only ordering guarantee is that the signal operation happens-after when `vkSignalSemaphore` is called and happens-before it returns. Therefore, it is invalid to call `vkSignalSemaphore` while there are any outstanding signal operations on that semaphore from any queue submissions unless those queue submissions have some dependency which ensures that they happen-after the host signal operation. One example of this would be if the pending signal operation is, itself, waiting on the same semaphore at a lower value and the call to `vkSignalSemaphore` signals that lower value. Furthermore, if there are two or more processes or threads signaling the same timeline semaphore from the host, the application must ensure that the `vkSignalSemaphore` with the lower semaphore value returns before `vkSignalSemaphore` is called with the higher value.

### 7.3. Fences

Fences are a synchronization primitive that can be used to insert a dependency from a queue to the host. Fences have two states - signaled and unsignaled. A fence can be signaled as part of the execution of a queue submission command. Fences can be unsignaled on the host with `vkResetFences`. Fences can be waited on by the host with the `vkWaitForFences` command, and the current state can be queried with `vkGetFenceStatus`.

The internal data of a fence may include a reference to any resources and pending work associated with signal or unsignal operations performed on that fence object, collectively referred to as the fence’s payload. Mechanisms to import and export that internal data to and from fences are provided below. These mechanisms indirectly enable applications to share fence state between two or more fences and other synchronization primitives across process and API boundaries.

Fences are represented by `VkFence` handles:
To create a fence, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateFence(
    VkDevice                                    device,
    const VkFenceCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFence*                                    pFence);
```

- `device` is the logical device that creates the fence.
- `pCreateInfo` is a pointer to a `VkFenceCreateInfo` structure containing information about how the fence is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pFence` is a pointer to a handle in which the resulting fence object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateFence-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkCreateFence-pCreateInfo-parameter**
  - `pCreateInfo` must be a valid pointer to a valid `VkFenceCreateInfo` structure
- **VUID-vkCreateFence-pAllocator-parameter**
  - If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateFence-pFence-parameter**
  - `pFence` must be a valid pointer to a `VkFence` handle

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkFenceCreateInfo` structure is defined as:
typedef struct VkFenceCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkFenceCreateFlags flags;
} VkFenceCreateInfo;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is a bitmask of **VkFenceCreateFlagBits** specifying the initial state and behavior of the fence.

### Valid Usage (Implicit)

- **VUID-VkFenceCreateInfo-sType-sType**
  sType **must** be **VK_STRUCTURE_TYPE_FENCE_CREATE_INFO**

- **VUID-VkFenceCreateInfo-pNext-pNext**
  Each pNext member of any structure (including this one) in the pNext chain **must** be either **NULL** or a pointer to a valid instance of **VkExportFenceCreateInfo** or **VkExportFenceWin32HandleInfoKHR**

- **VUID-VkFenceCreateInfo-sType-unique**
  The sType value of each struct in the pNext chain **must** be unique

- **VUID-VkFenceCreateInfo-flags-parameter**
  flags **must** be a valid combination of **VkFenceCreateFlagBits** values

---

typedef enum VkFenceCreateFlagBits {
    VK_FENCE_CREATE_SIGNALED_BIT = 0x00000001,
} VkFenceCreateFlagBits;

- **VK_FENCE_CREATE_SIGNALED_BIT** specifies that the fence object is created in the signaled state. Otherwise, it is created in the unsignaled state.

typedef VkFlags VkFenceCreateFlags;

**VkFenceCreateFlags** is a bitmask type for setting a mask of zero or more **VkFenceCreateFlagBits**.

To create a fence whose payload **can** be exported to external handles, add a **VkExportFenceCreateInfo** structure to the pNext chain of the **VkFenceCreateInfo** structure. The **VkExportFenceCreateInfo** structure is defined as:
### Valid Usage

- **VUID-VkExportFenceCreateInfo-handleTypes-01446**
  
  The bits in `handleTypes` must be supported and compatible, as reported by `VkExternalFenceProperties`.

### Valid Usage (Implicit)

- **VUID-VkExportFenceCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO`.

- **VUID-VkExportFenceCreateInfo-handleTypes-parameter**
  
  `handleTypes` must be a valid combination of `VkExternalFenceHandleTypeFlagBits` values.

To specify additional attributes of NT handles exported from a fence, add a `VkExportFenceWin32HandleInfoKHR` structure to the `pNext` chain of the `VkFenceCreateInfo` structure. The `VkExportFenceWin32HandleInfoKHR` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExportFenceCreateInfo {  
    VkStructureType                   sType;  
    const void*                       pNext;  
    VkExternalFenceHandleTypeFlags    handleTypes;  
} VkExportFenceCreateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence
typedef VkExportFenceCreateInfo VkExportFenceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **handleTypes** is a bitmask of `VkExternalFenceHandleTypeFlagBits` specifying one or more fence handle types the application can export from the resulting fence. The application can request multiple handle types for the same fence.
typedef struct VkExportFenceWin32HandleInfoKHR {
    VkStructureType               sType;
    const void*                   pNext;
    const SECURITY_ATTRIBUTES*    pAttributes;
    DWORD                         dwAccess;
    LPCWSTR                       name;
} VkExportFenceWin32HandleInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pAttributes** is a pointer to a Windows SECURITY_ATTRIBUTES structure specifying security attributes of the handle.
- **dwAccess** is a DWORD specifying access rights of the handle.
- **name** is a null-terminated UTF-16 string to associate with the underlying synchronization primitive referenced by NT handles exported from the created fence.

If **VkExportFenceCreateInfo** is not present in the same **pNext** chain, this structure is ignored.

If **VkExportFenceCreateInfo** is present in the **pNext** chain of **VkFenceCreateInfo** with a Windows **handleType**, but either **VkExportFenceWin32HandleInfoKHR** is not present in the **pNext** chain, or if it is but **pAttributes** is set to NULL, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”. Further, if the structure is not present, the access rights will be

```plaintext
DXGI_SHARED_RESOURCE_READ | DXGI_SHARED_RESOURCE_WRITE
```

for handles of the following types:

**VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT**

1


---

**Valid Usage**

- **VUID-VkExportFenceWin32HandleInfoKHR-handleTypes-01447**

If **VkExportFenceCreateInfo::handleTypes** does not include **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT**, a **VkExportFenceWin32HandleInfoKHR** structure must not be included in the **pNext** chain of **VkFenceCreateInfo**
To export a Windows handle representing the state of a fence, call:

```c
// Provided by VK_KHR_external_fence_win32
VkResult vkGetFenceWin32HandleKHR(
    VkDevice                                device,
    const VkFenceGetWin32HandleInfoKHR*    pGetWin32HandleInfo,  
    HANDLE*                                 pHandle);
```

- **device** is the logical device that created the fence being exported.
- **pGetWin32HandleInfo** is a pointer to a `VkFenceGetWin32HandleInfoKHR` structure containing parameters of the export operation.
- **pHandle** will return the Windows handle representing the fence state.

For handle types defined as NT handles, the handles returned by `vkGetFenceWin32HandleKHR` are owned by the application. To avoid leaking resources, the application must release ownership of them using the `CloseHandle` system call when they are no longer needed.

Exporting a Windows handle from a fence may have side effects depending on the transference of the specified handle type, as described in Importing Fence Payloads.
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_TOO_MANY_OBJECTS
• VK_ERROR_OUT_OF_HOST_MEMORY

The `VkFenceGetWin32HandleInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_fence_win32
typedef struct VkFenceGetWin32HandleInfoKHR {
    VkStructureType                      sType;
    const void*                          pNext;
    VkFence                              fence;
    VkExternalFenceHandleTypeFlagBits    handleType;
} VkFenceGetWin32HandleInfoKHR;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `fence` is the fence from which state will be exported.
• `handleType` is the type of handle requested.

The properties of the handle returned depend on the value of `handleType`. See `VkExternalFenceHandleTypeFlagBits` for a description of the properties of the defined external fence handle types.
Valid Usage

- **VUID-VkFenceGetWin32HandleInfoKHR-handleType-01448**
  
  handleType must have been included in `VkExportFenceCreateInfo::handleTypes` when the fence’s current payload was created.

- **VUID-VkFenceGetWin32HandleInfoKHR-handleType-01449**
  
  If handleType is defined as an NT handle, `vkGetFenceWin32HandleKHR` must be called no more than once for each valid unique combination of fence and handleType.

- **VUID-VkFenceGetWin32HandleInfoKHR-fence-01450**
  
  fence must not currently have its payload replaced by an imported payload as described below in Importing Fence Payloads unless that imported payload’s handle type was included in `VkExternalFenceProperties::exportFromImportedHandleTypes` for handleType.

- **VUID-VkFenceGetWin32HandleInfoKHR-handleType-01451**
  
  If handleType refers to a handle type with copy payload transference semantics, fence must be signaled, or have an associated fence signal operation pending execution.

- **VUID-VkFenceGetWin32HandleInfoKHR-handleType-01452**
  
  handleType must be defined as an NT handle or a global share handle.

Valid Usage (Implicit)

- **VUID-VkFenceGetWin32HandleInfoKHR-sType-sType**
  
  sType must be `VK_STRUCTURE_TYPE_FENCE_GET_WIN32_HANDLE_INFO_KHR`.

- **VUID-VkFenceGetWin32HandleInfoKHR-pNext-pNext**
  
  pNext must be NULL.

- **VUID-VkFenceGetWin32HandleInfoKHR-fence-parameter**
  
  fence must be a valid `VkFence` handle.

- **VUID-VkFenceGetWin32HandleInfoKHR-handleType-parameter**
  
  handleType must be a valid `VkExternalFenceHandleTypeFlagBits` value.

To export a POSIX file descriptor representing the payload of a fence, call:

```c
// Provided by VK_KHR_external_fence_fd
VkResult vkGetFenceFdKHR(
    VkDevice                                    device,
    const VkFenceGetFdInfoKHR*                  pGetFdInfo,
    int*                                        pFd);
```

- **device** is the logical device that created the fence being exported.

- **pGetFdInfo** is a pointer to a `VkFenceGetFdInfoKHR` structure containing parameters of the export operation.

- **pFd** will return the file descriptor representing the fence payload.
Each call to `vkGetFenceFdKHR` **must** create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application **must** release ownership of the file descriptor when it is no longer needed.

**Note**
Ownership can be released in many ways. For example, the application can call `close()` on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a fence payload.

If `pGetFdInfo->handleType` is `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` and the fence is signaled at the time `vkGetFenceFdKHR` is called, `pFd` **may** return the value `-1` instead of a valid file descriptor.

Where supported by the operating system, the implementation **must** set the file descriptor to be closed automatically when an `execve` system call is made.

Exporting a file descriptor from a fence **may** have side effects depending on the transference of the specified handle type, as described in Importing Fence State.

### Valid Usage (Implicit)

- **VUID-vkGetFenceFdKHR-device-parameter**
  - `device` **must** be a valid `VkDevice` handle
- **VUID-vkGetFenceFdKHR-pGetFdInfo-parameter**
  - `pGetFdInfo` **must** be a valid pointer to a valid `VkFenceGetFdInfoKHR` structure
- **VUID-vkGetFenceFdKHR-pFd-parameter**
  - `pFd` **must** be a valid pointer to an `int` value

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkFenceGetFdInfoKHR` structure is defined as:
Provided by VK_KHR_external_fence_fd

typedef struct VkFenceGetFdInfoKHR {
    VkStructureType                      sType;
    const void*                          pNext;
    VkFence                              fence;
    VkExternalFenceHandleTypeFlagBits    handleType;
} VkFenceGetFdInfoKHR;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **fence** is the fence from which state will be exported.
- **handleType** is the type of handle requested.

The properties of the file descriptor returned depend on the value of `handleType`. See `VkExternalFenceHandleTypeFlagBits` for a description of the properties of the defined external fence handle types.

Valid Usage

- **VUID-VkFenceGetFdInfoKHR-handleType-01453**
  
  `handleType` must have been included in `VkExportFenceCreateInfo::handleTypes` when `fence`'s current payload was created

- **VUID-VkFenceGetFdInfoKHR-handleType-01454**
  
  If `handleType` refers to a handle type with copy payload transference semantics, `fence` must be signaled, or have an associated `fence signal operation` pending execution

- **VUID-VkFenceGetFdInfoKHR-fence-01455**
  
  `fence` must not currently have its payload replaced by an imported payload as described below in Importing Fence Payloads unless that imported payload's handle type was included in `VkExternalFenceProperties::exportFromImportedHandleTypes` for `handleType`

- **VUID-VkFenceGetFdInfoKHR-handleType-01456**
  
  `handleType` must be defined as a POSIX file descriptor handle
Valid Usage (Implicit)

- **VUID-VkFenceGetFdInfoKHR-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR`

- **VUID-VkFenceGetFdInfoKHR-pNext-pNext**
  
  `pNext` **must** be `NULL`

- **VUID-VkFenceGetFdInfoKHR-fence-parameter**
  
  `fence` **must** be a valid `VkFence` handle

- **VUID-VkFenceGetFdInfoKHR-handleType-parameter**
  
  `handleType` **must** be a valid `VkExternalFenceHandleTypeFlagBits` value

To destroy a fence, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyFence( 
    VkDevice device, 
    VkFence fence, 
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the fence.
- `fence` is the handle of the fence to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- **VUID-vkDestroyFence-fence-01120**
  
  All queue submission commands that refer to `fence` **must** have completed execution

- **VUID-vkDestroyFence-fence-01121**
  
  If `VkAllocationCallbacks` were provided when `fence` was created, a compatible set of callbacks **must** be provided here

- **VUID-vkDestroyFence-fence-01122**
  
  If no `VkAllocationCallbacks` were provided when `fence` was created, `pAllocator` **must** be `NULL`
Valid Usage (Implicit)

- **VUID-vkDestroyFence-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkDestroyFence-fence-parameter**
  
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be a valid `VkFence` handle

- **VUID-vkDestroyFence-pAllocator-parameter**
  
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkDestroyFence-fence-parent**
  
  If `fence` is a valid handle, it must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `fence` must be externally synchronized

To query the status of a fence from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetFenceStatus(
        VkDevice device,
        VkFence fence);
```

- `device` is the logical device that owns the fence.
- `fence` is the handle of the fence to query.

Upon success, `vkGetFenceStatus` returns the status of the fence object, with the following return codes:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUCCESS</td>
<td>The fence specified by <code>fence</code> is signaled.</td>
</tr>
<tr>
<td>VK_NOT_READY</td>
<td>The fence specified by <code>fence</code> is unsignaled.</td>
</tr>
<tr>
<td>VK_ERROR_DEVICE_LOST</td>
<td>The device has been lost. See Lost Device.</td>
</tr>
</tbody>
</table>

If a queue submission command is pending execution, then the value returned by this command may immediately be out of date.

If the device has been lost (see Lost Device), `vkGetFenceStatus` may return any of the above status codes.
codes. If the device has been lost and `vkGetFenceStatus` is called repeatedly, it will eventually return either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`.

### Valid Usage (Implicit)

- **VUID-vkGetFenceStatus-device-parameter**
  
  *device* must be a valid `VkDevice` handle

- **VUID-vkGetFenceStatus-fence-parameter**
  
  *fence* must be a valid `VkFence` handle

- **VUID-vkGetFenceStatus-fence-parent**
  
  *fence* must have been created, allocated, or retrieved from *device*

### Return Codes

**Success**

- `VK_SUCCESS`
- `VK_NOT_READY`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

To set the state of fences to unsignaled from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetFences(
    VkDevice device,                      // Logical device that owns the fences.
    uint32_t fenceCount,                 // Number of fences to reset.
    const VkFence* pFences);            // Pointer to array of fence handles to reset.
```

- *device* is the logical device that owns the fences.
- *fenceCount* is the number of fences to reset.
- *pFences* is a pointer to an array of fence handles to reset.

If any member of *pFences* currently has its payload imported with temporary permanence, that fence’s prior permanent payload is first restored. The remaining operations described therefore operate on the restored payload.

When `vkResetFences` is executed on the host, it defines a fence unsignal operation for each fence, which resets the fence to the unsignaled state.

If any member of *pFences* is already in the unsignaled state when `vkResetFences` is executed, then
vkResetFences has no effect on that fence.

### Valid Usage

- VUID-vkResetFences-pFences-01123
  
  Each element of pFences must not be currently associated with any queue command that has not yet completed execution on that queue

### Valid Usage (Implicit)

- VUID-vkResetFences-device-parameter
  
  device must be a valid VkDevice handle

- VUID-vkResetFences-pFences-parameter
  
  pFences must be a valid pointer to an array of fenceCount valid VkFence handles

- VUID-vkResetFences-fenceCount-arraylength
  
  fenceCount must be greater than 0

- VUID-vkResetFences-pFences-parent
  
  Each element of pFences must have been created, allocated, or retrieved from device

### Host Synchronization

- Host access to each member of pFences must be externally synchronized

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_DEVICE_MEMORY

When a fence is submitted to a queue as part of a queue submission command, it defines a memory dependency on the batches that were submitted as part of that command, and defines a fence signal operation which sets the fence to the signaled state.

The first synchronization scope includes every batch submitted in the same queue submission command. Fence signal operations that are defined by vkQueueSubmit additionally include in the first synchronization scope all commands that occur earlier in submission order. Fence signal operations that are defined by vkQueueSubmit or vkQueueBindSparse additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in signal operation order.

The second synchronization scope only includes the fence signal operation.
The first access scope includes all memory access performed by the device.

The second access scope is empty.

To wait for one or more fences to enter the signaled state on the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkWaitForFences(
    VkDevice device,                  // Provided by VK_VERSION_1_0
    uint32_t fenceCount,              // Provided by VK_VERSION_1_0
    const VkFence* pFences,           // Provided by VK_VERSION_1_0
    VkBool32 waitAll,                 // Provided by VK_VERSION_1_0
    uint64_t timeout);                // Provided by VK_VERSION_1_0
```

- `device` is the logical device that owns the fences.
- `fenceCount` is the number of fences to wait on.
- `pFences` is a pointer to an array of `fenceCount` fence handles.
- `waitAll` is the condition that must be satisfied to successfully unblock the wait. If `waitAll` is `VK_TRUE`, then the condition is that all fences in `pFences` are signaled. Otherwise, the condition is that at least one fence in `pFences` is signaled.
- `timeout` is the timeout period in units of nanoseconds. `timeout` is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If the condition is satisfied when `vkWaitForFences` is called, then `vkWaitForFences` returns immediately. If the condition is not satisfied at the time `vkWaitForFences` is called, then `vkWaitForFences` will block and wait until the condition is satisfied or the `timeout` has expired, whichever is sooner.

If `timeout` is zero, then `vkWaitForFences` does not wait, but simply returns the current state of the fences. `VK_TIMEOUT` will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the condition is satisfied before the `timeout` has expired, `vkWaitForFences` returns `VK_SUCCESS`. Otherwise, `vkWaitForFences` returns `VK_TIMEOUT` after the `timeout` has expired.

If device loss occurs (see Lost Device) before the timeout has expired, `vkWaitForFences` must return in finite time with either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`.

**Note**
While we guarantee that `vkWaitForFences` must return in finite time, no guarantees are made that it returns immediately upon device loss. However, the client can reasonably expect that the delay will be on the order of seconds and that calling `vkWaitForFences` will not result in a permanently (or seemingly permanently) dead process.
Valid Usage (Implicit)

• VUID-vkWaitForFences-device-parameter
device must be a valid VkDevice handle

• VUID-vkWaitForFences-pFences-parameter
pFences must be a valid pointer to an array of fenceCount valid VkFence handles

• VUID-vkWaitForFences-fenceCount-arraylength
fenceCount must be greater than 0

• VUID-vkWaitForFences-pFences-parent
Each element of pFences must have been created, allocated, or retrieved from device

Return Codes

Success

• VK_SUCCESS
• VK_TIMEOUT

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_DEVICE_LOST

An execution dependency is defined by waiting for a fence to become signaled, either via vkWaitForFences or by polling on vkGetFenceStatus.

The first synchronization scope includes only the fence signal operation.

The second synchronization scope includes the host operations of vkWaitForFences or vkGetFenceStatus indicating that the fence has become signaled.

Note

Signaling a fence and waiting on the host does not guarantee that the results of memory accesses will be visible to the host, as the access scope of a memory dependency defined by a fence only includes device access. A memory barrier or other memory dependency must be used to guarantee this. See the description of host access types for more information.

7.3.1. Alternate Methods to Signal Fences

Besides submitting a fence to a queue as part of a queue submission command, a fence may also be signaled when a particular event occurs on a device or display.

To create a fence that will be signaled when an event occurs on a device, call:
Provided by VK_EXT_display_control

```c
VkResult vkRegisterDeviceEventEXT(
    VkDevice                                    device,
    const VkDeviceEventInfoEXT*                 pDeviceEventInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFence*                                    pFence);
```

- **device** is a logical device on which the event may occur.
- **pDeviceEventInfo** is a pointer to a VkDeviceEventInfoEXT structure describing the event of interest to the application.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pFence** is a pointer to a handle in which the resulting fence object is returned.

### Valid Usage (Implicit)

- **VUID-vkRegisterDeviceEventEXT-device-parameter**
  - device must be a valid VkDevice handle
- **VUID-vkRegisterDeviceEventEXT-pDeviceEventInfo-parameter**
  - pDeviceEventInfo must be a valid pointer to a valid VkDeviceEventInfoEXT structure
- **VUID-vkRegisterDeviceEventEXT-pAllocator-parameter**
  - If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
- **VUID-vkRegisterDeviceEventEXT-pFence-parameter**
  - pFence must be a valid pointer to a VkFence handle

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**

The VkDeviceEventInfoEXT structure is defined as:

```c
// Provided by VK_EXT_display_control
typedef struct VkDeviceEventInfoEXT {
    VkStructureType         sType;
    const void*             pNext;
    VkDeviceEventTypeEXT    deviceEvent;
} VkDeviceEventInfoEXT;
```
• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **device** is a VkDeviceEventTypeEXT value specifying when the fence will be signaled.

### Valid Usage (Implicit)

- VUID-VkDeviceEventInfoEXT-sType-sType
  
  **sType** must be VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT

- VUID-VkDeviceEventInfoEXT-pNext-pNext
  
  **pNext** must be NULL

- VUID-VkDeviceEventInfoEXT-deviceEvent-parameter
  
  **deviceEvent** must be a valid VkDeviceEventTypeEXT value

Possible values of **VkDeviceEventInfoEXT::device**, specifying when a fence will be signaled, are:

```c
// Provided by VK_EXT_display_control
typedef enum VkDeviceEventTypeEXT {
    VKDEVICE_EVENTTYPE_DISPLAYHOTPLUG_EXT = 0,
} VkDeviceEventTypeEXT;
```

- **VK_DEVICE_EVENT_TYPE_DISPLAY_HOTPLUG_EXT** specifies that the fence is signaled when a display is plugged into or unplugged from the specified device. Applications can use this notification to determine when they need to re-enumerate the available displays on a device.

To create a fence that will be signaled when an event occurs on a VkDisplayKHR object, call:

```c
// Provided by VK_EXT_display_control
VkResult vkRegisterDisplayEventEXT(
    VkDevice device,
    VkDisplayKHR display,
    const VkDisplayEventInfoEXT* pDisplayEventInfo,
    const VkAllocationCallbacks* pAllocator,
    VkFence* pFence);
```

- **device** is a logical device associated with **display**
- **display** is the display on which the event may occur.
- **pDisplayEventInfo** is a pointer to a VkDisplayEventInfoEXT structure describing the event of interest to the application.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pFence** is a pointer to a handle in which the resulting fence object is returned.
Valid Usage (Implicit)

- **VUID-vkRegisterDisplayEventEXT-device-parameter**
  *device* must be a valid *VkDevice* handle

- **VUID-vkRegisterDisplayEventEXT-display-parameter**
  *display* must be a valid *VkDisplayKHR* handle

- **VUID-vkRegisterDisplayEventEXT-pDisplayEventInfo-parameter**
  *pDisplayEventInfo* must be a valid pointer to a valid *VkDisplayEventInfoEXT* structure

- **VUID-vkRegisterDisplayEventEXT-pAllocator-parameter**
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkRegisterDisplayEventEXT-pFence-parameter**
  *pFence* must be a valid pointer to a *VkFence* handle

- **VUID-vkRegisterDisplayEventEXT-commonparent**
  Both of *device*, and *display* must have been created, allocated, or retrieved from the same *VkPhysicalDevice*

Return Codes

**Success**
- *VK_SUCCESS*

**Failure**
- *VK_ERROR_OUT_OF_HOST_MEMORY*

The *VkDisplayEventInfoEXT* structure is defined as:

```c
// Provided by VK_EXT_display_control
typedef struct VkDisplayEventInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDisplayEventTypeEXT displayEvent;
} VkDisplayEventInfoEXT;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *displayEvent* is a *VkDisplayEventTypeEXT* specifying when the fence will be signaled.
Valid Usage (Implicit)

- **VUID-VkDisplayEventInfoEXT-sType-sType**
  
  sType must be `VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT`

- **VUID-VkDisplayEventInfoEXT-pNext-pNext**
  
  pNext must be `NULL`

- **VUID-VkDisplayEventInfoEXT-displayEvent-parameter**
  
  displayEvent must be a valid `VkDisplayEventTypeEXT` value

Possible values of `VkDisplayEventInfoEXT::displayEvent`, specifying when a fence will be signaled, are:

```c
// Provided by VK_EXT_display_control
typedef enum VkDisplayEventTypeEXT {
    VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT = 0,
} VkDisplayEventTypeEXT;
```

- `VK_DISPLAY_EVENT_TYPE_FIRST_PIXEL_OUT_EXT` specifies that the fence is signaled when the first pixel of the next display refresh cycle leaves the display engine for the display.

### 7.3.2. Importing Fence Payloads

Applications can import a fence payload into an existing fence using an external fence handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the fence will be restored to its permanent state the next time that fence is passed to `vkResetFences`.

**Note**

Restoring a fence to its prior permanent payload is a distinct operation from resetting a fence payload. See `vkResetFences` for more detail.

Performing a subsequent temporary import on a fence before resetting it has no effect on this requirement; the next unsignal of the fence must still restore its last permanent state. A permanent payload import behaves as if the target fence was destroyed, and a new fence was created with the same handle but the imported payload. Because importing a fence payload temporarily or permanently detaches the existing payload from a fence, similar usage restrictions to those applied to `vkDestroyFence` are applied to any command that imports a fence payload. Which of these import types is used is referred to as the import operation’s permanence. Each handle type supports either one or both types of permanence.

The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external fence handle, depending on the handle’s type. The import method used is referred to as the handle type’s transference. When using handle types with reference transference, importing a payload to a fence adds the fence to the set of all fences sharing that payload. This set includes the fence from which the payload was exported. Fence signaling,
waiting, and resetting operations performed on any fence in the set must behave as if the set were a single fence. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Fence signaling, waiting, and resetting operations performed on the target of copy imports must not affect any other fence or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a fence payload to a handle with copy transference has the same side effects on the source fence's payload as executing a fence reset operation. If the fence was using a temporarily imported payload, the fence's prior permanent payload will be restored.

Note

The tables Handle Types Supported by VkImportFenceWin32HandleInfoKHR and Handle Types Supported by VkImportFenceFdInfoKHR define the permanence and transference of each handle type.

External synchronization allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for fences sharing a payload across processes, satisfying the external synchronization requirements of VkFence parameters as if all fences in the set were the same object is sometimes infeasible. Satisfying valid usage constraints on the state of a fence would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific fence handle, not to its payload. For distinct fence objects which share a payload:

- If multiple commands which queue a signal operation, or which unsignal a fence, are called concurrently, behavior will be as if the commands were called in an arbitrary sequential order.
- If a queue submission command is called with a fence that is sharing a payload, and the payload is already associated with another queue command that has not yet completed execution, either one or both of the commands will cause the fence to become signaled when they complete execution.
- If a fence payload is reset while it is associated with a queue command that has not yet completed execution, the payload will become unsignaled, but may become signaled again when the command completes execution.
- In the preceding cases, any of the devices associated with the fences sharing the payload may be lost, or any of the queue submission or fence reset commands may return VK_ERROR_INITIALIZATION_FAILED.

Other than these non-deterministic results, behavior is well defined. In particular:

- The implementation must not crash or enter an internally inconsistent state where future valid Vulkan commands might cause undefined results,
- Timeouts on future wait commands on fences sharing the payload must be effective.
These rules allow processes to synchronize access to shared memory without trusting each other. However, such processes must still be cautious not to use the shared fence for more than synchronizing access to the shared memory. For example, a process should not use a fence with shared payload to tell when commands it submitted to a queue have completed and objects used by those commands may be destroyed, since the other process can accidentally or maliciously cause the fence to signal before the commands actually complete.

When a fence is using an imported payload, its `VkExportFenceCreateInfo::handleTypes` value is that specified when creating the fence from which the payload was exported, rather than that specified when creating the fence. Additionally, `VkExternalFenceProperties::exportFromImportedHandleTypes` restricts which handle types can be exported from such a fence based on the specific handle type used to import the current payload. Passing a fence to `vkAcquireNextImageKHR` is equivalent to temporarily importing a fence payload to that fence.

Because the exportable handle types of an imported fence correspond to its current imported payload, and `vkAcquireNextImageKHR` behaves the same as a temporary import operation for which the source fence is opaque to the application, applications have no way of determining whether any external handle types can be exported from a fence in this state. Therefore, applications must not attempt to export handles from fences using a temporarily imported payload from `vkAcquireNextImageKHR`.

When importing a fence payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations must perform sufficient validation of external handles to ensure that the operation results in a valid fence which will not cause program termination, device loss, queue stalls, host thread stalls, or corruption of other resources when used as allowed according to its import parameters. If the external handle provided does not meet these requirements, the implementation must fail the fence payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

To import a fence payload from a Windows handle, call:

```c
// Provided by VK_KHR_external_fence_win32
VkResult vkImportFenceWin32HandleKHR(
    VkDevice device,
    const VkImportFenceWin32HandleInfoKHR* pImportFenceWin32HandleInfo);
```

- `device` is the logical device that created the fence.
- `pImportFenceWin32HandleInfo` is a pointer to a `VkImportFenceWin32HandleInfoKHR` structure specifying the fence and import parameters.

Importing a fence payload from Windows handles does not transfer ownership of the handle to the Vulkan implementation. For handle types defined as NT handles, the application must release...
ownership using the \texttt{CloseHandle} system call when the handle is no longer needed.

Applications can import the same fence payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

### Valid Usage

- \texttt{VUID-vkImportFenceWin32HandleKHR-fence-04448}
  - \texttt{fence} must not be associated with any queue command that has not yet completed execution on that queue

### Valid Usage (Implicit)

- \texttt{VUID-vkImportFenceWin32HandleKHR-device-parameter}
  - device must be a valid \texttt{VkDevice} handle
- \texttt{VUID-vkImportFenceWin32HandleKHR-pImportFenceWin32HandleInfo-parameter}
  - \texttt{pImportFenceWin32HandleInfo} must be a valid pointer to a valid \texttt{VkImportFenceWin32HandleInfoKHR} structure

### Return Codes

**Success**
- \texttt{VK_SUCCESS}

**Failure**
- \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
- \texttt{VK_ERROR_INVALID_EXTERNAL_HANDLE}

The \texttt{VkImportFenceWin32HandleInfoKHR} structure is defined as:

```c
// Provided by VK_KHR_external_fence_win32
typedef struct VkImportFenceWin32HandleInfoKHR {
    VkStructureType                      sType;
    const void*                          pNext;
    VkFence                              fence;
    VkFenceImportFlags                   flags;
    VkExternalFenceHandleTypeFlagBits    handleType;
    HANDLE                               handle;
    LPCWSTR                              name;
} VkImportFenceWin32HandleInfoKHR;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is NULL or a pointer to a structure extending this structure.
• **fence** is the fence into which the state will be imported.

• **flags** is a bitmask of `VkFenceImportFlagBits` specifying additional parameters for the fence payload import operation.

• **handleType** specifies the type of **handle**.

• **handle** is the external handle to import, or **NULL**.

• **name** is a null-terminated UTF-16 string naming the underlying synchronization primitive to import, or **NULL**.

The handle types supported by **handleType** are:

**Table 6. Handle Types Supported by `VkImportFenceWin32HandleInfoKHR`**

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT</code></td>
<td>Reference</td>
<td>Temporary, Permanent</td>
</tr>
<tr>
<td><code>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</code></td>
<td>Reference</td>
<td>Temporary, Permanent</td>
</tr>
</tbody>
</table>

**Valid Usage**

- **VUID-VkImportFenceWin32HandleInfoKHR-handleType-01457**
  
  **handleType** **must** be a value included in the Handle Types Supported by `VkImportFenceWin32HandleInfoKHR` table

- **VUID-VkImportFenceWin32HandleInfoKHR-handleType-01459**
  
  If **handleType** is not `VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT`, **name** **must** be **NULL**

- **VUID-VkImportFenceWin32HandleInfoKHR-handleType-01460**
  
  If **handle** is **NULL**, **name** **must** name a valid synchronization primitive of the type specified by **handleType**

- **VUID-VkImportFenceWin32HandleInfoKHR-handleType-01461**
  
  If **name** is **NULL**, **handle** **must** be a valid handle of the type specified by **handleType**

- **VUID-VkImportFenceWin32HandleInfoKHR-handle-01462**
  
  If **handle** is **not** **NULL**, **name** **must** be **NULL**

- **VUID-VkImportFenceWin32HandleInfoKHR-handle-01539**
  
  If **handle** is **not** **NULL**, it **must** obey any requirements listed for **handleType** in external fence handle types compatibility

- **VUID-VkImportFenceWin32HandleInfoKHR-name-01540**
  
  If **name** is **not** **NULL**, it **must** obey any requirements listed for **handleType** in external fence handle types compatibility
Valid Usage (Implicit)

- VUID-VkImportFenceWin32HandleInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMPORT_FENCE_WIN32_HANDLE_INFO_KHR

- VUID-VkImportFenceWin32HandleInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkImportFenceWin32HandleInfoKHR-fence-parameter
  fence must be a valid VkFence handle

- VUID-VkImportFenceWin32HandleInfoKHR-flags-parameter
  flags must be a valid combination of VkFenceImportFlagBits values

Host Synchronization

- Host access to fence must be externally synchronized

To import a fence payload from a POSIX file descriptor, call:

```c
// Provided by VK_KHR_external_fence_fd
VkResult vkImportFenceFdKHR(
    VkDevice                                    device,
    const VkImportFenceFdInfoKHR*               pImportFenceFdInfo);
```

- device is the logical device that created the fence.

- pImportFenceFdInfo is a pointer to a VkImportFenceFdInfoKHR structure specifying the fence and import parameters.

Importing a fence payload from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application must not perform any operations on the file descriptor after a successful import.

Applications can import the same fence payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

Valid Usage

- VUID-vkImportFenceFdKHR-fence-01463
  fence must not be associated with any queue command that has not yet completed execution on that queue
Valid Usage (Implicit)

- VUID-vkImportFenceFdKHR-device-parameter
  
  device must be a valid VkDevice handle

- VUID-vkImportFenceFdKHR-pImportFenceFdInfo-parameter
  
  pImportFenceFdInfo must be a valid pointer to a valid VkImportFenceFdInfoKHR structure

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkImportFenceFdInfoKHR structure is defined as:

```c
// Provided by VK_KHR_external_fence_fd
typedef struct VkImportFenceFdInfoKHR {
    VkStructureType                      sType;
    const void*                          pNext;
    VkFence                              fence;
    VkFenceImportFlags                   flags;
    VkExternalFenceHandleTypeFlagBits    handleType;
    int                                  fd;
} VkImportFenceFdInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- fence is the fence into which the payload will be imported.
- flags is a bitmask of VkFenceImportFlagBits specifying additional parameters for the fence payload import operation.
- handleType specifies the type of fd.
- fd is the external handle to import.

The handle types supported by handleType are:

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_FENCEHANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Reference</td>
<td>Temporary,Permanent</td>
</tr>
</tbody>
</table>

Chapter 7. Synchronization and Cache Control | 257
### Valid Usage

- **VUID-VkImportFenceFdInfoKHR-handleType-01464**
  
  **handleType must** be a value included in the Handle Types Supported by VkImportFenceFdInfoKHR table

- **VUID-VkImportFenceFdInfoKHR-fd-01541**
  
  **fd must** obey any requirements listed for **handleType** in external fence handle types compatibility

If **handleType** is **VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT**, the special value **-1** for **fd** is treated like a valid sync file descriptor referring to an object that has already signaled. The import operation will succeed and the **VkFence** will have a temporarily imported payload as if a valid file descriptor had been provided.

#### Note

This special behavior for importing an invalid sync file descriptor allows easier interoperability with other system APIs which use the convention that an invalid sync file descriptor represents work that has already completed and does not need to be waited for. It is consistent with the option for implementations to return a **-1** file descriptor when exporting a **VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT** from a **VkFence** which is signaled.

### Valid Usage (Implicit)

- **VUID-VkImportFenceFdInfoKHR-sType-sType**
  
  **sType must** be **VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR**

- **VUID-VkImportFenceFdInfoKHR-pNext-pNext**
  
  **pNext must** be **NULL**

- **VUID-VkImportFenceFdInfoKHR-fence-parameter**
  
  **fence must** be a valid **VkFence** handle

- **VUID-VkImportFenceFdInfoKHR-flags-parameter**
  
  **flags must** be a valid combination of **VkFenceImportFlagBits** values

- **VUID-VkImportFenceFdInfoKHR-handleType-parameter**
  
  **handleType must** be a valid **VkExternalFenceHandleTypeFlagBits** value

### Host Synchronization

- Host access to **fence must** be externally synchronized
Bits which can be set in `VkImportFenceWin32HandleInfoKHR::flags` and `VkImportFenceFdInfoKHR::flags` specifying additional parameters of a fence import operation are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkFenceImportFlagBits {
    VK_FENCE_IMPORT_TEMPORARY_BIT = 0x00000001,
} VkFenceImportFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence
typedef VkFenceImportFlagBits VkFenceImportFlagBitsKHR;
```

- `VK_FENCE_IMPORT_TEMPORARY_BIT` specifies that the fence payload will be imported only temporarily, as described in Importing Fence Payloads, regardless of the permanence of `handleType`.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkFenceImportFlags;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence
typedef VkFenceImportFlags VkFenceImportFlagsKHR;
```

`VkFenceImportFlags` is a bitmask type for setting a mask of zero or more `VkFenceImportFlagBits`.

### 7.4. Semaphores

Semaphores are a synchronization primitive that can be used to insert a dependency between queue operations or between a queue operation and the host. Binary semaphores have two states—signaled and unsignaled. Timeline semaphores have a strictly increasing 64-bit unsigned integer payload and are signaled with respect to a particular reference value. A semaphore can be signaled after execution of a queue operation is completed, and a queue operation can wait for a semaphore to become signaled before it begins execution. A timeline semaphore can additionally be signaled from the host with the `vkSignalSemaphore` command and waited on from the host with the `vkWaitSemaphores` command.

The internal data of a semaphore may include a reference to any resources and pending work associated with signal or unsignal operations performed on that semaphore object, collectively referred to as the semaphore's payload. Mechanisms to import and export that internal data to and from semaphores are provided below. These mechanisms indirectly enable applications to share semaphore state between two or more semaphores and other synchronization primitives across
process and API boundaries.

Semaphores are represented by `VkSemaphore` handles:

```
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphore)
```

To create a semaphore, call:

```
// Provided by VK_VERSION_1_0
VkResult vkCreateSemaphore(  
    VkDevice                                    device,  
    const VkSemaphoreCreateInfo*                pCreateInfo,  
    const VkAllocationCallbacks*                pAllocator,  
    VkSemaphore*                                pSemaphore);
```

- `device` is the logical device that creates the semaphore.
- `pCreateInfo` is a pointer to a `VkSemaphoreCreateInfo` structure containing information about how the semaphore is to be created.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSemaphore` is a pointer to a handle in which the resulting semaphore object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateSemaphore-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkCreateSemaphore-pCreateInfo-parameter**
  
  `pCreateInfo` must be a valid pointer to a valid `VkSemaphoreCreateInfo` structure

- **VUID-vkCreateSemaphore-pAllocator-parameter**
  
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateSemaphore-pSemaphore-parameter**
  
  `pSemaphore` must be a valid pointer to a `VkSemaphore` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
The `VkSemaphoreCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSemaphoreCreateInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkSemaphoreCreateFlags    flags;
} VkSemaphoreCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.

### Valid Usage (Implicit)

- `VUID-VkSemaphoreCreateInfo-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO`

- `VUID-VkSemaphoreCreateInfo-pNext-pNext`  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkExportSemaphoreCreateInfo`, `VkExportSemaphoreWin32HandleInfoKHR`, or `VkSemaphoreTypeCreateInfo`

- `VUID-VkSemaphoreCreateInfo-sType-unique`  
  The `sType` value of each struct in the `pNext` chain must be unique

- `VUID-VkSemaphoreCreateInfo-flags-zerobitmask`  
  `flags` must be 0

---

// Provided by VK_VERSION_1_0
```
typedef VkFlags VkSemaphoreCreateFlags;
```

`VkSemaphoreCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To create a semaphore of a specific type, add a `VkSemaphoreTypeCreateInfo` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkSemaphoreTypeCreateInfo` structure is defined as:

```c
typedef struct VkSemaphoreTypeCreateInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkSemaphoreType    semaphoreType;
    uint64_t           initialValue;
} VkSemaphoreTypeCreateInfo;
```

or the equivalent
// Provided by VK_KHR_timeline_semaphore
typedef VkSemaphoreTypeCreateInfo VkSemaphoreTypeCreateInfoKHR;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `semaphoreType` is a `VkSemaphoreType` value specifying the type of the semaphore.
- `initialValue` is the initial payload value if `semaphoreType` is `VK_SEMAPHORE_TYPE_TIMELINE`.

### Valid Usage (Implicit)

- VUID-VkSemaphoreTypeCreateInfo-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO`

- VUID-VkSemaphoreTypeCreateInfo-semaphoreType-parameter
  
  `semaphoreType` must be a valid `VkSemaphoreType`

### Valid Usage

- VUID-VkSemaphoreTypeCreateInfo-timelineSemaphore-03252
  
  If the `timelineSemaphore` feature is not enabled, `semaphoreType` must not equal `VK_SEMAPHORE_TYPE_TIMELINE`

- VUID-VkSemaphoreTypeCreateInfo-semaphoreType-03279
  
  If `semaphoreType` is `VK_SEMAPHORE_TYPE_BINARY`, `initialValue` must be zero

If no `VkSemaphoreTypeCreateInfo` structure is included in the `pNext` chain of `VkSemaphoreCreateInfo`, then the created semaphore will have a default `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY`.

Possible values of `VkSemaphoreTypeCreateInfo::semaphoreType`, specifying the type of a semaphore, are:

```c
typedef enum VkSemaphoreType {
    VK_SEMAPHORE_TYPE_BINARY = 0,
    VK_SEMAPHORE_TYPE_TIMELINE = 1,
    // Provided by VK_KHR_timeline_semaphore
    VK_SEMAPHORE_TYPE_BINARY_KHR = VK_SEMAPHORE_TYPE_BINARY,
    // Provided by VK_KHR_timeline_semaphore
    VK_SEMAPHORE_TYPE_TIMELINE_KHR = VK_SEMAPHORE_TYPE_TIMELINE,
} VkSemaphoreType;
```

or the equivalent

```c
// Provided by VK_KHR_timeline_semaphore
typedef VkSemaphoreType VkSemaphoreTypeKHR;
```
• **VK_SEMAPHORE_TYPE_BINARY** specifies a *binary semaphore* type that has a boolean payload indicating whether the semaphore is currently signaled or unsignaled. When created, the semaphore is in the unsignaled state.

• **VK_SEMAPHORE_TYPE_TIMELINE** specifies a *timeline semaphore* type that has a strictly increasing 64-bit unsigned integer payload indicating whether the semaphore is signaled with respect to a particular reference value. When created, the semaphore payload has the value given by the `initialValue` field of `VkSemaphoreTypeCreateInfo`.

To create a semaphore whose payload can be exported to external handles, add a `VkExportSemaphoreCreateInfo` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkExportSemaphoreCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExportSemaphoreCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalSemaphoreHandleTypeFlags handleTypes;
} VkExportSemaphoreCreateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_semaphore
typedef VkExportSemaphoreCreateInfo VkExportSemaphoreCreateInfoKHR;
```

• `sType` is the type of this structure.

• `pNext` is NULL or a pointer to a structure extending this structure.

• `handleTypes` is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying one or more semaphore handle types the application can export from the resulting semaphore. The application can request multiple handle types for the same semaphore.

---

**Valid Usage**

- VUID-VkExportSemaphoreCreateInfo-handleTypes-01124
  The bits in `handleTypes` must be supported and compatible, as reported by `VkExternalSemaphoreProperties`

---

**Valid Usage (Implicit)**

- VUID-VkExportSemaphoreCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO`

- VUID-VkExportSemaphoreCreateInfo-handleTypes-parameter
  `handleTypes` must be a valid combination of `VkExternalSemaphoreHandleTypeFlagBits` values
To specify additional attributes of NT handles exported from a semaphore, add a `VkExportSemaphoreWin32HandleInfoKHR` structure to the `pNext` chain of the `VkSemaphoreCreateInfo` structure. The `VkExportSemaphoreWin32HandleInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_win32
typedef struct VkExportSemaphoreWin32HandleInfoKHR {
    VkStructureType               sType;
    const void*                   pNext;
    const SECURITY_ATTRIBUTES*    pAttributes;
    DWORD                         dwAccess;
    LPCWSTR                       name;
} VkExportSemaphoreWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pAttributes` is a pointer to a Windows `SECURITY_ATTRIBUTES` structure specifying security attributes of the handle.
- `dwAccess` is a `DWORD` specifying access rights of the handle.
- `name` is a null-terminated UTF-16 string to associate with the underlying synchronization primitive referenced by NT handles exported from the created semaphore.

If `VkExportSemaphoreCreateInfo` is not present in the same `pNext` chain, this structure is ignored.

If `VkExportSemaphoreCreateInfo` is present in the `pNext` chain of `VkSemaphoreCreateInfo` with a Windows `handleType`, but either `VkExportSemaphoreWin32HandleInfoKHR` is not present in the `pNext` chain, or if it is but `pAttributes` is set to `NULL`, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”. Further, if the structure is not present, the access rights used depend on the handle type.

For handles of the following types:

**VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT**

The implementation **must** ensure the access rights allow both signal and wait operations on the semaphore.

For handles of the following types:

**VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT**

The access rights **must** be:

**GENERIC_ALL**

1

Valid Usage

- VUID-VkExportSemaphoreWin32HandleInfoKHR-handleTypes-01125
  If \( \text{VkExportSemaphoreCreateInfo::handleTypes} \) does not include \( \text{VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT} \) or \( \text{VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT} \), \( \text{VkExportSemaphoreWin32HandleInfoKHR} \) must not be included in the \( \text{pNext} \) chain of \( \text{VkSemaphoreCreateInfo} \).

Valid Usage (Implicit)

- VUID-VkExportSemaphoreWin32HandleInfoKHR-sType-sType
  \( \text{sType} \) must be \( \text{VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR} \)

- VUID-VkExportSemaphoreWin32HandleInfoKHR-pAttributes-parameter
  If \( \text{pAttributes} \) is not NULL, \( \text{pAttributes} \) must be a valid pointer to a valid \( \text{SECURITY_ATTRIBUTES} \) value.

To export a Windows handle representing the payload of a semaphore, call:

```c
// Provided by VK_KHR_external_semaphore_win32
VkResult vkGetSemaphoreWin32HandleKHR(
    VkDevice device,
    const VkSemaphoreGetWin32HandleInfoKHR* pGetWin32HandleInfo,
    HANDLE* pHandle);
```

- \( \text{device} \) is the logical device that created the semaphore being exported.
- \( \text{pGetWin32HandleInfo} \) is a pointer to a \( \text{VkSemaphoreGetWin32HandleInfoKHR} \) structure containing parameters of the export operation.
- \( \text{pHandle} \) will return the Windows handle representing the semaphore state.

For handle types defined as NT handles, the handles returned by \( \text{vkGetSemaphoreWin32HandleKHR} \) are owned by the application. To avoid leaking resources, the application must release ownership of them using the \text{CloseHandle} system call when they are no longer needed.

Exporting a Windows handle from a semaphore may have side effects depending on the transference of the specified handle type, as described in Importing Semaphore Payloads.
Valid Usage (Implicit)

- **VUID-vkGetSemaphoreWin32HandleKHR-device-parameter**
  
  *device must be a valid VkDevice handle*

- **VUID-vkGetSemaphoreWin32HandleKHR-pGetWin32HandleInfo-parameter**
  
  *pGetWin32HandleInfo must be a valid pointer to a valid VkSemaphoreGetWin32HandleInfoKHR structure*

- **VUID-vkGetSemaphoreWin32HandleKHR-pHandle-parameter**
  
  *pHandle must be a valid pointer to a HANDLE value*

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The **VkSemaphoreGetWin32HandleInfoKHR** structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_win32
typedef struct VkSemaphoreGetWin32HandleInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkSemaphoreGetWin32HandleInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **semaphore** is the semaphore from which state will be exported.
- **handleType** is the type of handle requested.

The properties of the handle returned depend on the value of **handleType**. See **VkExternalSemaphoreHandleTypeFlagBits** for a description of the properties of the defined external semaphore handle types.
Valid Usage

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-handleType-01126**
  
  `handleType` must have been included in `VkExportSemaphoreCreateInfo::handleTypes` when the semaphore's current payload was created.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-handleType-01127**
  
  If `handleType` is defined as an NT handle, `vkGetSemaphoreWin32HandleKHR` must be called no more than once for each valid unique combination of `semaphore` and `handleType`.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-semaphore-01128**
  
  `semaphore` must not currently have its payload replaced by an imported payload as described below in Importing Semaphore Payloads unless that imported payload’s handle type was included in `VkExternalSemaphoreProperties::exportFromImportedHandleTypes` for `handleType`.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-handleType-01129**
  
  If `handleType` refers to a handle type with copy payload transference semantics, as defined below in Importing Semaphore Payloads, there must be no queue waiting on `semaphore`.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-handleType-01130**
  
  If `handleType` refers to a handle type with copy payload transference semantics, `semaphore` must be signaled, or have an associated semaphore signal operation pending execution.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-handleType-01131**
  
  `handleType` must be defined as an NT handle or a global share handle.

Valid Usage (Implicit)

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR`.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-pNext-pNext**
  
  `pNext` must be NULL.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-semaphore-parameter**
  
  `semaphore` must be a valid `VkSemaphore` handle.

- **VUID-VkSemaphoreGetWin32HandleInfoKHR-handleType-parameter**
  
  `handleType` must be a valid `VkExternalSemaphoreHandleTypeFlagBits` value.

To export a POSIX file descriptor representing the payload of a semaphore, call:

```c
// Provided by VK_KHR_external_semaphore_fd
VkResult vkGetSemaphoreFdKHR(VkDevice device, const VkSemaphoreGetFdInfoKHR* pGetFdInfo, int* pFd);
```

- `device` is the logical device that created the semaphore being exported.
• \textit{pGetFdInfo} is a pointer to a \texttt{VkSemaphoreGetFdInfoKHR} structure containing parameters of the export operation.

• \textit{pFd} will return the file descriptor representing the semaphore payload.

Each call to \texttt{vkGetSemaphoreFdKHR} \textbf{must} create a new file descriptor and transfer ownership of it to the application. To avoid leaking resources, the application \textbf{must} release ownership of the file descriptor when it is no longer needed.

\begin{quote}
\textit{Note}
Ownership can be released in many ways. For example, the application can call \texttt{close()} on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a semaphore payload.
\end{quote}

Where supported by the operating system, the implementation \textbf{must} set the file descriptor to be closed automatically when an \texttt{execve} system call is made.

Exporting a file descriptor from a semaphore \textbf{may} have side effects depending on the transference of the specified handle type, as described in \texttt{Importing Semaphore State}.

\begin{quote}
\textbf{Valid Usage (Implicit)}
\begin{itemize}
\item VUID-vkGetSemaphoreFdKHR-device-parameter
\hspace{1em} device \textbf{must} be a valid \texttt{VkDevice} handle
\item VUID-vkGetSemaphoreFdKHR-pGetFdInfo-parameter
\hspace{1em} \textit{pGetFdInfo} \textbf{must} be a valid pointer to a valid \texttt{VkSemaphoreGetFdInfoKHR} structure
\item VUID-vkGetSemaphoreFdKHR-pFd-parameter
\hspace{1em} \textit{pFd} \textbf{must} be a valid pointer to an \texttt{int} value
\end{itemize}
\end{quote}

\begin{quote}
\textbf{Return Codes}
\begin{itemize}
\item \textbf{Success}
\hspace{1em} \texttt{VK_SUCCESS}
\item \textbf{Failure}
\hspace{1em} \texttt{VK_ERROR_TOO_MANY_OBJECTS}
\hspace{1em} \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
\end{itemize}
\end{quote}

The \texttt{VkSemaphoreGetFdInfoKHR} structure is defined as:
// Provided by VK_KHR_external_semaphore_fd

typedef struct VkSemaphoreGetFdInfoKHR {
  VkStructureType                          sType;
  const void*                              pNext;
  VkSemaphore                              semaphore;
  VkExternalSemaphoreHandleTypeFlagBits    handleType;
} VkSemaphoreGetFdInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **semaphore** is the semaphore from which state will be exported.
- **handleType** is the type of handle requested.

The properties of the file descriptor returned depend on the value of **handleType**. See `VkExternalSemaphoreHandleTypeFlagBits` for a description of the properties of the defined external semaphore handle types.

**Valid Usage**

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-01132**
  handleType must have been included in `VkExportSemaphoreCreateInfo::handleTypes` when semaphore's current payload was created

- **VUID-VkSemaphoreGetFdInfoKHR-semaphore-01133**
  semaphore must not currently have its payload replaced by an imported payload as described below in Importing Semaphore Payloads unless that imported payload's handle type was included in `VkExternalSemaphoreProperties::exportFromImportedHandleTypes` for handleType

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-01134**
  If handleType refers to a handle type with copy payload transference semantics, as defined below in Importing Semaphore Payloads, there must be no queue waiting on semaphore

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-01135**
  If handleType refers to a handle type with copy payload transference semantics, semaphore must be signaled, or have an associated semaphore signal operation pending execution

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-01136**
  handleType must be defined as a POSIX file descriptor handle

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-03253**
  If handleType refers to a handle type with copy payload transference semantics, semaphore must have been created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-03254**
  If handleType refers to a handle type with copy payload transference semantics, semaphore must have an associated semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution
Valid Usage (Implicit)

- **VUID-VkSemaphoreGetFdInfoKHR-sType-sType**
  sType must be VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR

- **VUID-VkSemaphoreGetFdInfoKHR-pNext-pNext**
  pNext must be NULL

- **VUID-VkSemaphoreGetFdInfoKHR-semaphore-parameter**
  semaphore must be a valid VkSemaphore handle

- **VUID-VkSemaphoreGetFdInfoKHR-handleType-parameter**
  handleType must be a valid VkExternalSemaphoreHandleTypeFlagBits value

To export a Zircon event handle representing the payload of a semaphore, call:

```c
// Provided by VK_FUCHSIA_external_semaphore
VkResult vkGetSemaphoreZirconHandleFUCHSIA(
    VkDevice                                    device,
    const    VkSemaphoreGetZirconHandleInfoFUCHSIA*  pGetZirconHandleInfo,
    zx_handle_t*                                pZirconHandle);
```

- **device** is the logical device that created the semaphore being exported.
- **pGetZirconHandleInfo** is a pointer to a VkSemaphoreGetZirconHandleInfoFUCHSIA structure containing parameters of the export operation.
- **pZirconHandle** will return the Zircon event handle representing the semaphore payload.

Each call to **vkGetSemaphoreZirconHandleFUCHSIA** must create a Zircon event handle and transfer ownership of it to the application. To avoid leaking resources, the application must release ownership of the Zircon event handle when it is no longer needed.

**Note**

Ownership can be released in many ways. For example, the application can call zx_handle_close() on the file descriptor, or transfer ownership back to Vulkan by using the file descriptor to import a semaphore payload.

Exporting a Zircon event handle from a semaphore may have side effects depending on the transference of the specified handle type, as described in Importing Semaphore State.
Valid Usage (Implicit)

- VUID-vkGetSemaphoreZirconHandleFUCHSIA-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetSemaphoreZirconHandleFUCHSIA-pGetZirconHandleInfo-parameter
  pGetZirconHandleInfo must be a valid pointer to a valid VkSemaphoreGetZirconHandleInfoFUCHSIA structure

- VUID-vkGetSemaphoreZirconHandleFUCHSIA-pZirconHandle-parameter
  pZirconHandle must be a valid pointer to a zx_handle_t value

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The VkSemaphoreGetZirconHandleInfoFUCHSIA structure is defined as:

```c
typedef struct VkSemaphoreGetZirconHandleInfoFUCHSIA {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
} VkSemaphoreGetZirconHandleInfoFUCHSIA;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- semaphore is the semaphore from which state will be exported.
- handleType is the type of handle requested.

The properties of the Zircon event handle returned depend on the value of handleType. See VkExternalSemaphoreHandleTypeFlagBits for a description of the properties of the defined external semaphore handle types.
Valid Usage

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-handleType-04758**
  
  ```
  handleType must have been included in VkExportSemaphoreCreateInfo::handleTypes when_semaphore's current payload was created.
  ```

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-semaphore-04759**
  
  ```
  semaphore must not currently have its payload replaced by an imported payload as described below in Importing Semaphore Payloads unless that imported payload's handle type was included in VkExternalSemaphoreProperties::exportFromImportedHandleTypes for handleType.
  ```

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-handleType-04760**
  
  If handleType refers to a handle type with copy payload transference semantics, as defined below in Importing Semaphore Payloads, there must be no queue waiting on semaphore.

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-handleType-04761**
  
  If handleType refers to a handle type with copy payload transference semantics, semaphore must be signaled, or have an associated semaphore signal operation pending execution.

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-handleType-04762**
  
  handleType must be defined as a Zircon event handle.

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-semaphore-04763**
  
  ```
  semaphore must have been created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY.
  ```

Valid Usage (Implicit)

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-sType-sType**
  
  ```
  sType must be VK_STRUCTURE_TYPE_SEMAPHORE_GET_ZIRCON_HANDLE_INFO_FUCHSIA
  ```

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-pNext-pNext**
  
  ```
  pNext must be NULL
  ```

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-semaphore-parameter**
  
  ```
  semaphore must be a valid VkSemaphore handle
  ```

- **VUID-VkSemaphoreGetZirconHandleInfoFUCHSIA-handleType-parameter**
  
  ```
  handleType must be a valid VkExternalSemaphoreHandleTypeFlagBits value
  ```

To destroy a semaphore, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroySemaphore(  
    VkDevice device,  
    VkSemaphore semaphore,  
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the semaphore.
• semaphore is the handle of the semaphore to destroy.

• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroySemaphore-semaphore-01137
  All submitted batches that refer to semaphore must have completed execution

- VUID-vkDestroySemaphore-semaphore-01138
  If VkAllocationCallbacks were provided when semaphore was created, a compatible set of callbacks must be provided here

- VUID-vkDestroySemaphore-semaphore-01139
  If no VkAllocationCallbacks were provided when semaphore was created, pAllocator must be NULL

Valid Usage (Implicit)

- VUID-vkDestroySemaphore-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroySemaphore-semaphore-parameter
  If semaphore is not VK_NULL_HANDLE, semaphore must be a valid VkSemaphore handle

- VUID-vkDestroySemaphore-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroySemaphore-semaphore-parent
  If semaphore is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

• Host access to semaphore must be externally synchronized

7.4.1. Semaphore Signaling

When a batch is submitted to a queue via a queue submission, and it includes semaphores to be signaled, it defines a memory dependency on the batch, and defines semaphore signal operations which set the semaphores to the signaled state.

In case of semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE the semaphore is considered signaled with respect to the counter value set to be signaled as specified in VkTimelineSemaphoreSubmitInfo or VkSemaphoreSignalInfo.

The first synchronization scope includes every command submitted in the same batch. In the case of vkQueueSubmit2KHR, the first synchronization scope is limited to the pipeline stage specified by
VkSemaphoreSubmitInfoKHR::stage. Semaphore signal operations that are defined by vkQueueSubmit or vkQueueSubmit2KHR additionally include all commands that occur earlier in submission order. Semaphore signal operations that are defined by vkQueueSubmit or vkQueueBindSparse additionally include in the first synchronization scope any semaphore and fence signal operations that occur earlier in signal operation order.

The second synchronization scope includes only the semaphore signal operation.

The first access scope includes all memory access performed by the device.

The second access scope is empty.

### 7.4.2. Semaphore Waiting

When a batch is submitted to a queue via a queue submission, and it includes semaphores to be waited on, it defines a memory dependency between prior semaphore signal operations and the batch, and defines semaphore wait operations.

Such semaphore wait operations set the semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY to the unsignaled state. In case of semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE a prior semaphore signal operation defines a memory dependency with a semaphore wait operation if the value the semaphore is signaled with is greater than or equal to the value the semaphore is waited with, thus the semaphore will continue to be considered signaled with respect to the counter value waited on as specified in VkTimelineSemaphoreSubmitInfo.

The first synchronization scope includes all semaphore signal operations that operate on semaphores waited on in the same batch, and that happen-before the wait completes.

The second synchronization scope includes every command submitted in the same batch. In the case of vkQueueSubmit, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by the corresponding element of pWaitDstStageMask. In the case of vkQueueSubmit2KHR, the second synchronization scope is limited to the pipeline stage specified by VkSemaphoreSubmitInfoKHR::stage. Also, in the case of either vkQueueSubmit2KHR or vkQueueSubmit, the second synchronization scope additionally includes all commands that occur later in submission order.

The first access scope is empty.

The second access scope includes all memory access performed by the device.

The semaphore wait operation happens-after the first set of operations in the execution dependency, and happens-before the second set of operations in the execution dependency.
Note
Unlike timeline semaphores, fences or events, the act of waiting for a binary semaphore also unsignals that semaphore. Applications must ensure that between two such wait operations, the semaphore is signaled again, with execution dependencies used to ensure these occur in order. Binary semaphore waits and signals should thus occur in discrete 1:1 pairs.

Note
A common scenario for using pWaitDstStageMask with values other than VK_PIPELINE_STAGE_ALL_COMMANDS_BIT is when synchronizing a window system presentation operation against subsequent command buffers which render the next frame. In this case, a presentation image must not be overwritten until the presentation operation completes, but other pipeline stages can execute without waiting. A mask of VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT prevents subsequent color attachment writes from executing until the semaphore signals. Some implementations may be able to execute transfer operations and/or vertex processing work before the semaphore is signaled.

If an image layout transition needs to be performed on a presentable image before it is used in a framebuffer, that can be performed as the first operation submitted to the queue after acquiring the image, and should not prevent other work from overlapping with the presentation operation. For example, a VkImageMemoryBarrier could use:

- srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- srcAccessMask = 0
- dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT.
- oldLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
- newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

Alternatively, oldLayout can be VK_IMAGE_LAYOUT_UNDEFINED, if the image’s contents need not be preserved.

This barrier accomplishes a dependency chain between previous presentation operations and subsequent color attachment output operations, with the layout transition performed in between, and does not introduce a dependency between previous work and any vertex processing stages. More precisely, the semaphore signals after the presentation operation completes, the semaphore wait stalls the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT stage, and there is a dependency from that same stage to itself with the layout transition performed in between.
7.4.3. Semaphore State Requirements for Wait Operations

Before waiting on a semaphore, the application must ensure the semaphore is in a valid state for a wait operation. Specifically, when a semaphore wait operation is submitted to a queue:

- A binary semaphore must be signaled, or have an associated semaphore signal operation that is pending execution.
- Any semaphore signal operations on which the pending binary semaphore signal operation depends must also be completed or pending execution.
- There must be no other queue waiting on the same binary semaphore when the operation executes.

7.4.4. Host Operations on Semaphores

In addition to semaphore signal operations and semaphore wait operations submitted to device queues, timeline semaphores support the following host operations:

- Query the current counter value of the semaphore using the `vkGetSemaphoreCounterValue` command.
- Wait for a set of semaphores to reach particular counter values using the `vkWaitSemaphores` command.
- Signal the semaphore with a particular counter value from the host using the `vkSignalSemaphore` command.

To query the current counter value of a semaphore created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` from the host, call:

```
// Provided by VK_KHR_timeline_semaphore
VkResult vkGetSemaphoreCounterValueKHR(
    VkDevice                                    device,  // Provided by VK_KHR_timeline_semaphore
    VkSemaphore                                 semaphore,  // Provided by VK_KHR_timeline_semaphore
    uint64_t*                                   pValue);  // Provided by VK_KHR_timeline_semaphore
```

- `device` is the logical device that owns the semaphore.
- `semaphore` is the handle of the semaphore to query.
- `pValue` is a pointer to a 64-bit integer value in which the current counter value of the semaphore is returned.

**Note**

If a queue submission command is pending execution, then the value returned by this command may immediately be out of date.
**Valid Usage**

- VUID-vkGetSemaphoreCounterValue-semaphore-03255
  
  ```
  semaphore must have been created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE
  ```

**Valid Usage (Implicit)**

- VUID-vkGetSemaphoreCounterValue-device-parameter
  
  ```
  device must be a valid VkDevice handle
  ```

- VUID-vkGetSemaphoreCounterValue-semaphore-parameter
  
  ```
  semaphore must be a valid VkSemaphore handle
  ```

- VUID-vkGetSemaphoreCounterValue-pValue-parameter
  
  ```
  pValue must be a valid pointer to a uint64_t value
  ```

- VUID-vkGetSemaphoreCounterValue-semaphore-parent
  
  ```
  semaphore must have been created, allocated, or retrieved from device
  ```

**Return Codes**

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

To wait for a set of semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE to reach particular counter values on the host, call:

```c
// Provided by VK_KHR_timeline_semaphore
VkResult vkWaitSemaphoresKHR(
    VkDevice                                    device,
    const VkSemaphoreWaitInfo*                  pWaitInfo,
    uint64_t                                    timeout);
```

- **device** is the logical device that owns the semaphore.
- **pWaitInfo** is a pointer to a VkSemaphoreWaitInfo structure containing information about the wait condition.
- **timeout** is the timeout period in units of nanoseconds. timeout is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer...
than one nanosecond, and may be longer than the requested period.

If the condition is satisfied when `vkWaitSemaphores` is called, then `vkWaitSemaphores` returns immediately. If the condition is not satisfied at the time `vkWaitSemaphores` is called, then `vkWaitSemaphores` will block and wait until the condition is satisfied or the `timeout` has expired, whichever is sooner.

If `timeout` is zero, then `vkWaitSemaphores` does not wait, but simply returns information about the current state of the semaphore. `VK_TIMEOUT` will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the condition is satisfied before the `timeout` has expired, `vkWaitSemaphores` returns `VK_SUCCESS`. Otherwise, `vkWaitSemaphores` returns `VK_TIMEOUT` after the `timeout` has expired.

If device loss occurs (see Lost Device) before the timeout has expired, `vkWaitSemaphores` must return in finite time with either `VK_SUCCESS` or `VK_ERROR_DEVICE_LOST`.

### Valid Usage (Implicit)

- VUID-vkWaitSemaphores-device-parameter
  - `device` must be a valid `VkDevice` handle

- VUID-vkWaitSemaphores-pWaitInfo-parameter
  - `pWaitInfo` must be a valid pointer to a valid `VkSemaphoreWaitInfo` structure

### Return Codes

**Success**

- `VK_SUCCESS`
- `VK_TIMEOUT`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`

The `VkSemaphoreWaitInfo` structure is defined as:
typedef struct VkSemaphoreWaitInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkSemaphoreWaitFlags    flags;
    uint32_t                semaphoreCount;
    const VkSemaphore*      pSemaphores;
    const uint64_t*         pValues;
} VkSemaphoreWaitInfo;

or the equivalent

// Provided by VK_KHR_timeline_semaphore
typedef VkSemaphoreWaitInfo VkSemaphoreWaitInfoKHR;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is a bitmask of VkSemaphoreWaitFlagBits specifying additional parameters for the semaphore wait operation.
• **semaphoreCount** is the number of semaphores to wait on.
• **pSemaphores** is a pointer to an array of semaphoreCount semaphore handles to wait on.
• **pValues** is a pointer to an array of semaphoreCount timeline semaphore values.

### Valid Usage

• **VUID-VkSemaphoreWaitInfo-pSemaphores-03256**
  
  All of the elements of **pSemaphores** must reference a semaphore that was created with a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_TIMELINE**
Valid Usage (Implicit)

- **VUID-VkSemaphoreWaitInfo-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO`

- **VUID-VkSemaphoreWaitInfo-pNext-pNext**
  
  `pNext` **must** be `NULL`

- **VUID-VkSemaphoreWaitInfo-flags-parameter**
  
  `flags` **must** be a valid combination of `VkSemaphoreWaitFlagBits` values

- **VUID-VkSemaphoreWaitInfo-pSemaphores-parameter**
  
  `pSemaphores` **must** be a valid pointer to an array of `semaphoreCount` valid `VkSemaphore` handles

- **VUID-VkSemaphoreWaitInfo-pValues-parameter**
  
  `pValues` **must** be a valid pointer to an array of `semaphoreCount` `uint64_t` values

- **VUID-VkSemaphoreWaitInfo-semaphoreCount-arraylength**
  
  `semaphoreCount` **must** be greater than `0`

Bits which **can** be set in `VkSemaphoreWaitInfo::flags`, specifying additional parameters of a semaphore wait operation, are:

```c
typedef enum VkSemaphoreWaitFlagBits {
    VK_SEMAPHORE_WAIT_ANY_BIT = 0x00000001,
    // Provided by VK_KHR_timeline_semaphore
    VK_SEMAPHORE_WAIT_ANY_BIT_KHR = VK_SEMAPHORE_WAIT_ANY_BIT,
} VkSemaphoreWaitFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_timeline_semaphore
typedef VkSemaphoreWaitFlagBits VkSemaphoreWaitFlagBitsKHR;
```

- `VK_SEMAPHORE_WAIT_ANY_BIT` specifies that the semaphore wait condition is that at least one of the semaphores in `VkSemaphoreWaitInfo::pSemaphores` has reached the value specified by the corresponding element of `VkSemaphoreWaitInfo::pValues`. If `VK_SEMAPHORE_WAIT_ANY_BIT` is not set, the semaphore wait condition is that all of the semaphores in `VkSemaphoreWaitInfo::pSemaphores` have reached the value specified by the corresponding element of `VkSemaphoreWaitInfo::pValues`.

```c
typedef VkFlags VkSemaphoreWaitFlags;
```

or the equivalent

```
```
VkSemaphoreWaitFlags is a bitmask type for setting a mask of zero or more VkSemaphoreWaitFlagBits.

To signal a semaphore created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE with a particular counter value, on the host, call:

```c
VkResult vkSignalSemaphoreKHR(VkDevice device, const VkSemaphoreSignalInfo* pSignalInfo);
```

- `device` is the logical device that owns the semaphore.
- `pSignalInfo` is a pointer to a VkSemaphoreSignalInfo structure containing information about the signal operation.

When `vkSignalSemaphore` is executed on the host, it defines and immediately executes a semaphore signal operation which sets the timeline semaphore to the given value.

The first synchronization scope is defined by the host execution model, but includes execution of `vkSignalSemaphore` on the host and anything that happened-before it.

The second synchronization scope is empty.

### Valid Usage (Implicit)

- VUID-vkSignalSemaphore-device-parameter
  - `device` must be a valid VkDevice handle
- VUID-vkSignalSemaphore-pSignalInfo-parameter
  - `pSignalInfo` must be a valid pointer to a valid VkSemaphoreSignalInfo structure

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OFDEVICE_MEMORY

The VkSemaphoreSignalInfo structure is defined as:
```c
typedef struct VkSemaphoreSignalInfo {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    uint64_t value;
} VkSemaphoreSignalInfo;
```

or the equivalent

```c
// Provided by VK_KHR_timeline_semaphore
typedef VkSemaphoreSignalInfo VkSemaphoreSignalInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **semaphore** is the handle of the semaphore to signal.
- **value** is the value to signal.

**Valid Usage**

- VUID-VkSemaphoreSignalInfo-semaphore-03257
  `semaphore` must have been created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`

- VUID-VkSemaphoreSignalInfo-value-03258
  `value` must have a value greater than the current value of the semaphore

- VUID-VkSemaphoreSignalInfo-value-03259
  `value` must be less than the value of any pending semaphore signal operations

- VUID-VkSemaphoreSignalInfo-value-03260
  `value` must have a value which does not differ from the current value of the semaphore or the value of any outstanding semaphore wait or signal operation on `semaphore` by more than `maxTimelineSemaphoreValueDifference`

**Valid Usage (Implicit)**

- VUID-VkSemaphoreSignalInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO`

- VUID-VkSemaphoreSignalInfo-pNext-pNext
  `pNext` must be **NULL**

- VUID-VkSemaphoreSignalInfo-semaphore-parameter
  `semaphore` must be a valid `VkSemaphore` handle
## 7.4.5. Importing Semaphore Payloads

Applications can import a semaphore payload into an existing semaphore using an external semaphore handle. The effects of the import operation will be either temporary or permanent, as specified by the application. If the import is temporary, the implementation must restore the semaphore to its prior permanent state after submitting the next semaphore wait operation. Performing a subsequent temporary import on a semaphore before performing a semaphore wait has no effect on this requirement; the next wait submitted on the semaphore must still restore its last permanent state. A permanent payload import behaves as if the target semaphore was destroyed, and a new semaphore was created with the same handle but the imported payload. Because importing a semaphore payload temporarily or permanently detaches the existing payload from a semaphore, similar usage restrictions to those applied to `vkDestroySemaphore` are applied to any command that imports a semaphore payload. Which of these import types is used is referred to as the import operation's permanence. Each handle type supports either one or both types of permanence.

The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external semaphore handle, depending on the handle's type. The import method used is referred to as the handle type's transference. When using handle types with reference transference, importing a payload to a semaphore adds the semaphore to the set of all semaphores sharing that payload. This set includes the semaphore from which the payload was exported. Semaphoresignaling and waiting operations performed on any semaphore in the set must behave as if the set were a single semaphore. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Semaphoresignaling and waiting operations performed on the target of copy imports must not affect any other semaphore or payload.

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a semaphore payload to a handle with copy transference has the same side effects on the source semaphore's payload as executing a semaphore wait operation. If the semaphore was using a temporarily imported payload, the semaphore's prior permanent payload will be restored.

### Note
The permanence and transference of handle types can be found in:

- **Handle Types Supported by** `VkImportSemaphoreWin32HandleInfoKHR`
- **Handle Types Supported by** `VkImportSemaphoreFdInfoKHR`
- **Handle Types Supported by** `VkImportSemaphoreZirconHandleInfoFUCHSIA`

**External synchronization** allows implementations to modify an object's internal state, i.e. payload, without internal synchronization. However, for semaphores sharing a payload across processes, satisfying the external synchronization requirements of `VkSemaphore` parameters as if all semaphores in the set were the same object is sometimes infeasible. Satisfying the wait operation state requirements would similarly require impractical coordination or levels of trust between processes. Therefore, these constraints only apply to a specific semaphore handle, not to its payload. For distinct semaphore objects which share a payload, if the semaphores are passed to
separate queue submission commands concurrently, behavior will be as if the commands were called in an arbitrary sequential order. If the wait operation state requirements are violated for the shared payload by a queue submission command, or if a signal operation is queued for a shared payload that is already signaled or has a pending signal operation, effects must be limited to one or more of the following:

- Returning VK_ERROR_INITIALIZATION_FAILED from the command which resulted in the violation.
- Losing the logical device on which the violation occurred immediately or at a future time, resulting in a VK_ERROR_DEVICE_LOST error from subsequent commands, including the one causing the violation.
- Continuing execution of the violating command or operation as if the semaphore wait completed successfully after an implementation-dependent timeout. In this case, the state of the payload becomes undefined, and future operations on semaphores sharing the payload will be subject to these same rules. The semaphore must be destroyed or have its payload replaced by an import operation to again have a well-defined state.

Note
These rules allow processes to synchronize access to shared memory without trusting each other. However, such processes must still be cautious not to use the shared semaphore for more than synchronizing access to the shared memory. For example, a process should not use a shared semaphore as part of an execution dependency chain that, when complete, leads to objects being destroyed, if it does not trust other processes sharing the semaphore payload.

When a semaphore is using an imported payload, its VkExportSemaphoreCreateInfo::handleTypes value is that specified when creating the semaphore from which the payload was exported, rather than that specified when creating the semaphore. Additionally, VkExternalSemaphoreProperties::exportFromImportedHandleTypes restricts which handle types can be exported from such a semaphore based on the specific handle type used to import the current payload. Passing a semaphore to vkAcquireNextImageKHR is equivalent to temporarily importing a semaphore payload to that semaphore.

Note
Because the exportable handle types of an imported semaphore correspond to its current imported payload, and vkAcquireNextImageKHR behaves the same as a temporary import operation for which the source semaphore is opaque to the application, applications have no way of determining whether any external handle types can be exported from a semaphore in this state. Therefore, applications must not attempt to export external handles from semaphores using a temporarily imported payload from vkAcquireNextImageKHR.

When importing a semaphore payload, it is the responsibility of the application to ensure the external handles meet all valid usage requirements. However, implementations must perform sufficient validation of external handles to ensure that the operation results in a valid semaphore which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its import parameters, and excepting those side effects allowed for violations of the valid semaphore state for wait operations rules. If the external
handle provided does not meet these requirements, the implementation must fail the semaphore payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

In addition, when importing a semaphore payload that is not compatible with the payload type corresponding to the `VkSemaphoreType` the semaphore was created with, the implementation may fail the semaphore payload import operation with the error code `VK_ERROR_INVALID_EXTERNAL_HANDLE`.

### Note

As the introduction of the external semaphore handle type `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT` predates that of timeline semaphores, support for importing semaphore payloads from external handles of that type into semaphores created (implicitly or explicitly) with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` is preserved for backwards compatibility. However, applications should prefer importing such handle types into semaphores created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE`.

To import a semaphore payload from a Windows handle, call:

```c
// Provided by VK_KHR_external_semaphore_win32
VkResult vkImportSemaphoreWin32HandleKHR(
    VkDevice                                    device,
    const VkImportSemaphoreWin32HandleInfoKHR*  pImportSemaphoreWin32HandleInfo);
```

- `device` is the logical device that created the semaphore.
- `pImportSemaphoreWin32HandleInfo` is a pointer to a `VkImportSemaphoreWin32HandleInfoKHR` structure specifying the semaphore and import parameters.

Importing a semaphore payload from Windows handles does not transfer ownership of the handle to the Vulkan implementation. For handle types defined as NT handles, the application must release ownership using the `CloseHandle` system call when the handle is no longer needed.

Applications can import the same semaphore payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

### Valid Usage (Implicit)

- VUID-vkImportSemaphoreWin32HandleKHR-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkImportSemaphoreWin32HandleKHR-pImportSemaphoreWin32HandleInfo-parameter
  - `pImportSemaphoreWin32HandleInfo` must be a valid pointer to a valid `VkImportSemaphoreWin32HandleInfoKHR` structure
Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The `VkImportSemaphoreWin32HandleInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_win32
typedef struct VkImportSemaphoreWin32HandleInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkSemaphoreImportFlags flags;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
    HANDLE handle;
    LPCWSTR name;
} VkImportSemaphoreWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `semaphore` is the semaphore into which the payload will be imported.
- `flags` is a bitmask of `VkSemaphoreImportFlagBits` specifying additional parameters for the semaphore payload import operation.
- `handleType` specifies the type of `handle`.
- `handle` is the external handle to import, or `NULL`.
- `name` is a null-terminated UTF-16 string naming the underlying synchronization primitive to import, or `NULL`.

The handle types supported by `handleType` are:

<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Reference</td>
<td>Temporary,Permanent</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Reference</td>
<td>Temporary,Permanent</td>
</tr>
</tbody>
</table>

Table 8. Handle Types Supported by `VkImportSemaphoreWin32HandleInfoKHR`

286 | Chapter 7. Synchronization and Cache Control
<table>
<thead>
<tr>
<th>Handle Type</th>
<th>Transference</th>
<th>Permanence Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT</td>
<td>Reference</td>
<td>Temporary,Permanent</td>
</tr>
</tbody>
</table>

### Valid Usage

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-01140**
  - `handleType` **must** be a value included in the **Handle Types Supported by VkImportSemaphoreWin32HandleInfoKHR** table

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-01466**
  - If `handleType` is not VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT or VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT, `name` **must** be NULL

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-01467**
  - If `handle` is NULL, `name` **must** name a valid synchronization primitive of the type specified by `handleType`

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-01468**
  - If `name` is NULL, `handle` **must** be a valid handle of the type specified by `handleType`

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-01469**
  - If `handle` is not NULL, `name` **must** be NULL

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-01542**
  - If `handle` is not NULL, it **must** obey any requirements listed for `handleType` in **external semaphore handle types compatibility**

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-name-01543**
  - If `name` is not NULL, it **must** obey any requirements listed for `handleType` in **external semaphore handle types compatibility**

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-03261**
  - If `handleType` is VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT or VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT, the VkSemaphoreCreateInfo::flags field **must** match that of the semaphore from which `handle` or `name` was exported

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-handleType-03262**
  - If `handleType` is VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT or VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT, the VkSemaphoreTypeCreateInfo::semaphoreType field **must** match that of the semaphore from which `handle` or `name` was exported

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-flags-03322**
  - If `flags` contains VK_SEMAPHORE_IMPORT_TEMPORARY_BIT, the VkSemaphoreTypeCreateInfo::semaphoreType field of the semaphore from which `handle` or `name` was exported **must** not be VK_SEMAPHORE_TYPE_TIMELINE
Valid Usage (Implicit)

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-sType-sType**
  
  *sType must be* `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-pNext-pNext**
  
  *pNext must be NULL*

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-semaphore-parameter**
  
  *semaphore must be a valid* `VkSemaphore` *handle*

- **VUID-VkImportSemaphoreWin32HandleInfoKHR-flags-parameter**
  
  *flags must be a valid combination of* `VkSemaphoreImportFlagBits` *values*

Host Synchronization

- Host access to *semaphore must be externally synchronized*

To import a semaphore payload from a POSIX file descriptor, call:

```c
// Provided by VK_KHR_external_semaphore_fd
VkResult vkImportSemaphoreFdKHR(
    VkDevice device,
    const VkImportSemaphoreFdInfoKHR* pImportSemaphoreFdInfo);
```

- **device** is the logical device that created the semaphore.

- **pImportSemaphoreFdInfo** is a pointer to a `VkImportSemaphoreFdInfoKHR` structure specifying the semaphore and import parameters.

Importing a semaphore payload from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application **must** not perform any operations on the file descriptor after a successful import.

Applications **can** import the same semaphore payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

Valid Usage

- **VUID-vkImportSemaphoreFdKHR-semaphore-01142**
  
  *semaphore must not be associated with any queue command that has not yet completed execution on that queue*
Valid Usage (Implicit)

- VUID-vkImportSemaphoreFdKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkImportSemaphoreFdKHR-pImportSemaphoreFdInfo-parameter
  pImportSemaphoreFdInfo must be a valid pointer to a valid VkImportSemaphoreFdInfoKHR structure

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkImportSemaphoreFdInfoKHR structure is defined as:

```c
// Provided by VK_KHR_external_semaphore_fd
typedef struct VkImportSemaphoreFdInfoKHR {
    VkStructureType                          sType;
    const void*                              pNext;
    VkSemaphore                              semaphore;
    VkSemaphoreImportFlags                   flags;
    VkExternalSemaphoreHandleTypeFlagBits    handleType;
    int                                       fd;
} VkImportSemaphoreFdInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **semaphore** is the semaphore into which the payload will be imported.
- **flags** is a bitmask of VkSemaphoreImportFlagBits specifying additional parameters for the semaphore payload import operation.
- **handleType** specifies the type of fd.
- **fd** is the external handle to import.

The handle types supported by handleType are:

Table 9. Handle Types Supported by VkImportSemaphoreFdInfoKHR
### Valid Usage

- **VUID-VkImportSemaphoreFdInfoKHR-handleType-01143**
  
  `handleType` **must** be a value included in the `Handle Types Supported by VkImportSemaphoreFdInfoKHR` table.

- **VUID-VkImportSemaphoreFdInfoKHR-fd-01544**
  
  `fd` **must** obey any requirements listed for `handleType` in external semaphore handle types compatibility.

- **VUID-VkImportSemaphoreFdInfoKHR-handleType-03263**
  
  If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`, the `VkSemaphoreCreateInfo::flags` field **must** match that of the semaphore from which `fd` was exported.

- **VUID-VkImportSemaphoreFdInfoKHR-handleType-03264**
  
  If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT`, the `VkSemaphoreTypeCreateInfo::semaphoreType` field **must** match that of the semaphore from which `fd` was exported.

- **VUID-VkImportSemaphoreFdInfoKHR-flags-03323**
  
  If `flags` contains `VK_SEMAPHORE_IMPORT_TEMPORARY_BIT`, the `VkSemaphoreTypeCreateInfo::semaphoreType` field of the semaphore from which `fd` was exported **must** not be `VK_SEMAPHORE_TYPE_TIMELINE`.

---

If `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT`, the special value `-1` for `fd` is treated like a valid sync file descriptor referring to an object that has already signaled. The import operation will succeed and the `VkSemaphore` will have a temporarily imported payload as if a valid file descriptor had been provided.

**Note**

This special behavior for importing an invalid sync file descriptor allows easier interoperability with other system APIs which use the convention that an invalid sync file descriptor represents work that has already completed and does not need to be waited for. It is consistent with the option for implementations to return a `-1` file descriptor when exporting a `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT` from a `VkSemaphore` which is signaled.
Valid Usage (Implicit)

- **VUID-VkImportSemaphoreFdInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR`

- **VUID-VkImportSemaphoreFdInfoKHR-pNext-pNext**
  
  `pNext` must be `NULL`

- **VUID-VkImportSemaphoreFdInfoKHR-semaphore-parameter**
  
  `semaphore` must be a valid `VkSemaphore` handle

- **VUID-VkImportSemaphoreFdInfoKHR-flags-parameter**
  
  `flags` must be a valid combination of `VkSemaphoreImportFlagBits` values

- **VUID-VkImportSemaphoreFdInfoKHR-handleType-parameter**
  
  `handleType` must be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

Host Synchronization

- Host access to `semaphore` must be externally synchronized

To import a semaphore payload from a Zircon event handle, call:

```c
// Provided by VK_FUCHSIA_external_semaphore
VkResult vkImportSemaphoreZirconHandleFUCHSIA(
    VkDevice device,
    const VkImportSemaphoreZirconHandleInfoFUCHSIA* pImportSemaphoreZirconHandleInfo);
```

- `device` is the logical device that created the semaphore.
- `pImportSemaphoreZirconHandleInfo` is a pointer to a `VkImportSemaphoreZirconHandleInfoFUCHSIA` structure specifying the semaphore and import parameters.

Importing a semaphore payload from a Zircon event handle transfers ownership of the handle from the application to the Vulkan implementation. The application must not perform any operations on the handle after a successful import.

Applications can import the same semaphore payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance.

Valid Usage

- **VUID-vkImportSemaphoreZirconHandleFUCHSIA-semaphore-04764**
  
  `semaphore` must not be associated with any queue command that has not yet completed execution on that queue.
Valid Usage (Implicit)

- VUID-vkImportSemaphoreZirconHandleFUCHSIA-device-parameter
device must be a valid VkDevice handle

- VUID-vkImportSemaphoreZirconHandleFUCHSIA-pImportSemaphoreZirconHandleInfo-parameter
pImportSemaphoreZirconHandleInfo must be a valid pointer to a valid VkImportSemaphoreZirconHandleInfoFUCHSIA structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkImportSemaphoreZirconHandleInfoFUCHSIA structure is defined as:

```c
// Provided by VK_FUCHSIA_external_semaphore
typedef struct VkImportSemaphoreZirconHandleInfoFUCHSIA {
    VkStructureType sType;
    const void* pNext;
    VkSemaphore semaphore;
    VkSemaphoreImportFlags flags;
    VkExternalSemaphoreHandleTypeFlagBits handleType;
    zx_handle_t zirconHandle;
} VkImportSemaphoreZirconHandleInfoFUCHSIA;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- semaphore is the semaphore into which the payload will be imported.
- flags is a bitmask of VkSemaphoreImportFlagBits specifying additional parameters for the semaphore payload import operation.
- handleType specifies the type of zirconHandle.
- zirconHandle is the external handle to import.

The handle types supported by handleType are:

Table 10. Handle Types Supported by VkImportSemaphoreZirconHandleInfoFUCHSIA
### Valid Usage

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-handleType-04765**
  
  `handleType` **must** be a value included in the **Handle Types Supported by VkImportSemaphoreZirconHandleInfoFUCHSIA** table.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-zirconHandle-04766**
  
  `zirconHandle` **must** obey any requirements listed for `handleType` in **external semaphore handle types compatibility**.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-zirconHandle-04767**
  
  `zirconHandle` **must** have `ZX_RIGHTS_BASIC` and `ZX_RIGHTS_SIGNAL` rights.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-semaphoreType-04768**
  
  The `VkSemaphoreTypeCreateInfo::semaphoreType` field **must** not be `VK_SEMAPHORE_TYPE_TIMELINE`.

### Valid Usage (Implicit)

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_ZIRCON_HANDLE_INFO_FUCHSIA`.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-pNext-pNext**
  
  `pNext` **must** be `NULL`.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-semaphore-parameter**
  
  `semaphore` **must** be a valid `VkSemaphore` handle.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-flags-parameter**
  
  `flags` **must** be a valid combination of `VkSemaphoreImportFlagBits` values.

- **VUID-VkImportSemaphoreZirconHandleInfoFUCHSIA-handleType-parameter**
  
  `handleType` **must** be a valid `VkExternalSemaphoreHandleTypeFlagBits` value.

### Host Synchronization

- Host access to `semaphore` **must** be externally synchronized.

Additional parameters of a semaphore import operation are specified by `VkImportSemaphoreWin32HandleInfoKHR::flags` or `VkImportSemaphoreFdInfoKHR::flags`. Bits which can be set include:
or the equivalent

```
// Provided by VK_KHR_external_semaphore
typedef VkSemaphoreImportFlagBits VkSemaphoreImportFlagBitsKHR;
```

These bits have the following meanings:

- `VK_SEMAPHORE_IMPORT_TEMPORARY_BIT` specifies that the semaphore payload will be imported only temporarily, as described in Importing Semaphore Payloads, regardless of the permanence of `handleType`.

```
// Provided by VK_VERSION_1_1
typedef VkFlags VkSemaphoreImportFlags;
```

or the equivalent

```
// Provided by VK_KHR_external_semaphore
typedef VkSemaphoreImportFlags VkSemaphoreImportFlagsKHR;
```

`VkSemaphoreImportFlags` is a bitmask type for setting a mask of zero or more `VkSemaphoreImportFlagBits`.

### 7.5. Events

Events are a synchronization primitive that can be used to insert a fine-grained dependency between commands submitted to the same queue, or between the host and a queue. Events must not be used to insert a dependency between commands submitted to different queues. Events have two states - signaled and unsignaled. An application can signal or unsignal an event either on the host or on the device. A device can be made to wait for an event to become signaled before executing further operations. No command exists to wait for an event to become signaled on the host, but the current state of an event can be queried.

Events are represented by `VkEvent` handles:

```
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkEvent)
```
To create an event, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateEvent(
    VkDevice device,
    const VkEventCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkEvent* pEvent);
```

- **device** is the logical device that creates the event.
- **pCreateInfo** is a pointer to a `VkEventCreateInfo` structure containing information about how the event is to be created.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pEvent** is a pointer to a handle in which the resulting event object is returned.

When created, the event object is in the unsignaled state.

### Valid Usage

- **VUID-vkCreateEvent-events-04468**
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::events` is `VK_FALSE`, then the implementation does not support events, and `vkCreateEvent` must not be used.

### Valid Usage (Implicit)

- **VUID-vkCreateEvent-device-parameter**
  `device` must be a valid `VkDevice` handle
- **VUID-vkCreateEvent-pCreateInfo-parameter**
  `pCreateInfo` must be a valid pointer to a valid `VkEventCreateInfo` structure
- **VUID-vkCreateEvent-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateEvent-pEvent-parameter**
  `pEvent` must be a valid pointer to a `VkEvent` handle
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkEventCreateInfo` structure is defined as:

```cpp
// Provided by VK_VERSION_1_0
typedef struct VkEventCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkEventCreateFlags flags;
} VkEventCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkEventCreateFlagBits` defining additional creation parameters.

Valid Usage (Implicit)

- VUID-VkEventCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_EVENT_CREATE_INFO`
- VUID-VkEventCreateInfo-pNext-pNext
  `pNext` must be NULL
- VUID-VkEventCreateInfo-flags-parameter
  `flags` must be a valid combination of `VkEventCreateFlagBits` values

```cpp
// Provided by VK_VERSION_1_0
typedef VkFlags VkEventCreateFlags;
```

`VkEventCreateFlags` is a bitmask type for setting a mask of `VkEventCreateFlagBits`.

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkEventCreateFlagBits {
    // Provided by VK_KHR_synchronization2
    VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR = 0x00000001,
} VkEventCreateFlagBits;
```
VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR specifies that host event commands will not be used with this event.

To destroy an event, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyEvent(
    VkDevice device,
    VkEvent event,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the event.
- **event** is the handle of the event to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyEvent-event-01145**
  All submitted commands that refer to **event** must have completed execution

- **VUID-vkDestroyEvent-event-01146**
  If **VkAllocationCallbacks** were provided when **event** was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyEvent-event-01147**
  If no **VkAllocationCallbacks** were provided when **event** was created, **pAllocator** must be NULL

### Valid Usage (Implicit)

- **VUID-vkDestroyEvent-device-parameter**
  **device** must be a valid **VkDevice** handle

- **VUID-vkDestroyEvent-event-parameter**
  If **event** is not **VK_NULL_HANDLE**, **event** must be a valid **VkEvent** handle

- **VUID-vkDestroyEvent-pAllocator-parameter**
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- **VUID-vkDestroyEvent-event-parent**
  If **event** is a valid handle, it must have been created, allocated, or retrieved from **device**

### Host Synchronization

- Host access to **event** must be externally synchronized
To query the state of an event from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetEventStatus(
    VkDevice device,
    VkEvent event);
```

- `device` is the logical device that owns the event.
- `event` is the handle of the event to query.

Upon success, `vkGetEventStatus` returns the state of the event object with the following return codes:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EVENT_SET</td>
<td>The event specified by <code>event</code> is signaled.</td>
</tr>
<tr>
<td>VK_EVENT_RESET</td>
<td>The event specified by <code>event</code> is unsignaled.</td>
</tr>
</tbody>
</table>

If a `vkCmdSetEvent` or `vkCmdResetEvent` command is in a command buffer that is in the pending state, then the value returned by this command may immediately be out of date.

The state of an event can be updated by the host. The state of the event is immediately changed, and subsequent calls to `vkGetEventStatus` will return the new state. If an event is already in the requested state, then updating it to the same state has no effect.

### Valid Usage

- VUID-vkGetEventStatus-event-03940
  
  `event` must not have been created with `VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR`

### Valid Usage (Implicit)

- VUID-vkGetEventStatus-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkGetEventStatus-event-parameter
  
  `event` must be a valid `VkEvent` handle

- VUID-vkGetEventStatus-event-parent
  
  `event` must have been created, allocated, or retrieved from `device`
Return Codes

**Success**
- VK_EVENT_SET
- VK_EVENT_RESET

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

To set the state of an event to signaled from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkSetEvent(
    VkDevice device,
    VkEvent event);
```

- `device` is the logical device that owns the event.
- `event` is the event to set.

When `vkSetEvent` is executed on the host, it defines an *event signal operation* which sets the event to the signaled state.

If `event` is already in the signaled state when `vkSetEvent` is executed, then `vkSetEvent` has no effect, and no event signal operation occurs.

**Valid Usage**
- `VUID-vkSetEvent-event-03941`
  - `event` must not have been created with `VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR`

**Valid Usage (Implicit)**
- `VUID-vkSetEvent-device-parameter`
  - `device` must be a valid `VkDevice` handle
- `VUID-vkSetEvent-event-parameter`
  - `event` must be a valid `VkEvent` handle
- `VUID-vkSetEvent-event-parent`
  - `event` must have been created, allocated, or retrieved from `device`
Host Synchronization

- Host access to event must be externally synchronized

Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To set the state of an event to unsignaled from the host, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetEvent(
    VkDevice device,
    VkEvent event);
```

- `device` is the logical device that owns the event.
- `event` is the event to reset.

When `vkResetEvent` is executed on the host, it defines an event unsignal operation which resets the event to the unsignaled state.

If `event` is already in the unsignaled state when `vkResetEvent` is executed, then `vkResetEvent` has no effect, and no event unsignal operation occurs.

Valid Usage

- VUID-vkResetEvent-event-03821
  There must be an execution dependency between `vkCmdResetEvent` and the execution of any `vkCmdWaitEvents` that includes `event` in its `pEvents` parameter

- VUID-vkResetEvent-event-03822
  There must be an execution dependency between `vkCmdResetEvent` and the execution of any `vkCmdWaitEvents2KHR` that includes `event` in its `pEvents` parameter

- VUID-vkResetEvent-event-03823
  `event` must not have been created with VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR
Valid Usage (Implicit)

- **VUID-vkResetEvent-device-parameter**
  
  device must be a valid VkDevice handle

- **VUID-vkResetEvent-event-parameter**
  
  event must be a valid VkEvent handle

- **VUID-vkResetEvent-event-parent**
  
  event must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to event must be externally synchronized

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_DEVICE_MEMORY

The state of an event can also be updated on the device by commands inserted in command buffers.

To signal an event from a device, call:

```c
// Provided by VK_KHR_synchronization2
void vkCmdSetEvent2KHR(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    const VkDependencyInfoKHR* pDependencyInfo);
```

- commandBuffer is the command buffer into which the command is recorded.
- event is the event that will be signaled.
- pDependencyInfo is a pointer to a VkDependencyInfoKHR structure defining the first scopes of this operation.

When `vkCmdSetEvent2KHR` is submitted to a queue, it defines the first half of memory dependencies defined by `pDependencyInfo`, as well as an event signal operation which sets the event to the signaled state. A memory dependency is defined between the event signal operation and commands that occur earlier in submission order.

The first synchronization scope and access scope are defined by the union of all the memory
dependencies defined by \texttt{pDependencyInfo}, and are applied to all operations that occur earlier in submission order. Queue family ownership transfers and image layout transitions defined by \texttt{pDependencyInfo} are also included in the first scopes.

The second synchronization scope includes only the event signal operation, and any queue family ownership transfers and image layout transitions defined by \texttt{pDependencyInfo}.

The second access scope includes only queue family ownership transfers and image layout transitions.

Future \texttt{vkCmdWaitEvents2KHR} commands rely on all values of each element in \texttt{pDependencyInfo} matching exactly with those used to signal the corresponding event. \texttt{vkCmdWaitEvents} must not be used to wait on the result of a signal operation defined by \texttt{vkCmdSetEvent2KHR}.

\begin{note}

The extra information provided by \texttt{vkCmdSetEvent2KHR} compared to \texttt{vkCmdSetEvent} allows implementations to more efficiently schedule the operations required to satisfy the requested dependencies. With \texttt{vkCmdSetEvent}, the full dependency information is not known until \texttt{vkCmdWaitEvents} is recorded, forcing implementations to insert the required operations at that point and not before.

If \texttt{event} is already in the signaled state when \texttt{vkCmdSetEvent2KHR} is executed on the device, then \texttt{vkCmdSetEvent2KHR} has no effect, no event signal operation occurs, and no dependency is generated.

\end{note}

\textbf{Valid Usage}

- \textbf{VUID-vkCmdSetEvent2KHR-synchronization2-03824}
  
  The \textit{synchronization2} feature must be enabled

- \textbf{VUID-vkCmdSetEvent2KHR-dependencyFlags-03825}
  
  The \textit{dependencyFlags} member of \texttt{dependencyInfo} must be 0

- \textbf{VUID-vkCmdSetEvent2KHR-commandBuffer-03826}
  
  The current device mask of \texttt{commandBuffer} must include exactly one physical device.

- \textbf{VUID-vkCmdSetEvent2KHR-srcStageMask-03827}
  
  The \textit{srcStageMask} member of any element of the \texttt{pMemoryBarriers}, \texttt{pBufferMemoryBarriers}, or \texttt{pImageMemoryBarriers} members of \texttt{pDependencyInfo} must only include pipeline stages valid for the queue family that was used to create the command pool that \texttt{commandBuffer} was allocated from

- \textbf{VUID-vkCmdSetEvent2KHR-dstStageMask-03828}
  
  The \textit{dstStageMask} member of any element of the \texttt{pMemoryBarriers}, \texttt{pBufferMemoryBarriers}, or \texttt{pImageMemoryBarriers} members of \texttt{pDependencyInfo} must only include pipeline stages valid for the queue family that was used to create the command pool that \texttt{commandBuffer} was allocated from
Valid Usage (Implicit)

- VUID-vkCmdSetEvent2KHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdSetEvent2KHR-event-parameter
  event must be a valid VkEvent handle

- VUID-vkCmdSetEvent2KHR-pDependencyInfo-parameter
  pDependencyInfo must be a valid pointer to a valid VkDependencyInfoKHR structure

- VUID-vkCmdSetEvent2KHR-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdSetEvent2KHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdSetEvent2KHR-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdSetEvent2KHR-commonparent
  Both of commandBuffer, and event must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Outside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The VkDependencyInfoKHR structure is defined as:
typedef struct VkDependencyInfoKHR {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDependencyFlags                   dependencyFlags;
    uint32_t                            memoryBarrierCount;
    const VkMemoryBarrier2KHR*          pMemoryBarriers;
    uint32_t                            bufferMemoryBarrierCount;
    const VkBufferMemoryBarrier2KHR*    pBufferMemoryBarriers;
    uint32_t                            imageMemoryBarrierCount;
    const VkImageMemoryBarrier2KHR*     pImageMemoryBarriers;
} VkDependencyInfoKHR;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **dependencyFlags** is a bitmask of *VkDependencyFlagBits* specifying how execution and memory dependencies are formed.
• **memoryBarrierCount** is the length of the *pMemoryBarriers* array.
• **pMemoryBarriers** is a pointer to an array of *VkMemoryBarrier2KHR* structures that define memory dependencies between any memory accesses.
• **bufferMemoryBarrierCount** is the length of the *pBufferMemoryBarriers* array.
• **pBufferMemoryBarriers** is a pointer to an array of *VkBufferMemoryBarrier2KHR* structures that define memory dependencies between buffer ranges.
• **imageMemoryBarrierCount** is the length of the *pImageMemoryBarriers* array.
• **pImageMemoryBarriers** is a pointer to an array of *VkImageMemoryBarrier2KHR* structures that define memory dependencies between image subresources.

This structure defines a set of memory dependencies, as well as queue family transfer operations and image layout transitions.

Each member of *pMemoryBarriers*, *pBufferMemoryBarriers*, and *pImageMemoryBarriers* defines a separate memory dependency.
Valid Usage (Implicit)

- **VUID-VkDependencyInfoKHR-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_DEPENDENCY_INFO_KHR*

- **VUID-VkDependencyInfoKHR-pNext-pNext**
  
  *pNext must be NULL*

- **VUID-VkDependencyInfoKHR-dependencyFlags-parameter**
  
  *dependencyFlags must be a valid combination of VkDependencyFlagBits values*

- **VUID-VkDependencyInfoKHR-pMemoryBarriers-parameter**
  
  *If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of memoryBarrierCount valid VkMemoryBarrier2KHR structures*

- **VUID-VkDependencyInfoKHR-pBufferMemoryBarriers-parameter**
  
  *If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier2KHR structures*

- **VUID-VkDependencyInfoKHR-pImageMemoryBarriers-parameter**
  
  *If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an array of imageMemoryBarrierCount valid VkImageMemoryBarrier2KHR structures*

To set the state of an event to signaled from a device, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);
```

- **commandBuffer** is the command buffer into which the command is recorded.
- **event** is the event that will be signaled.
- **stageMask** specifies the source stage mask used to determine the first synchronization scope.

`vkCmdSetEvent` behaves identically to `vkCmdSetEvent2KHR`, except that it does not define an access scope, and must only be used with `vkCmdWaitEvents`, not `vkCmdWaitEvents2KHR`. 
Valid Usage

- VUID-vkCmdSetEvent-stageMask-04090
  If the geometry shaders feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- VUID-vkCmdSetEvent-stageMask-04091
  If the tessellation shaders feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- VUID-vkCmdSetEvent-stageMask-04092
  If the conditional rendering feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`

- VUID-vkCmdSetEvent-stageMask-04093
  If the fragment density map feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

- VUID-vkCmdSetEvent-stageMask-04094
  If the transform feedback feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`

- VUID-vkCmdSetEvent-stageMask-04095
  If the mesh shaders feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`

- VUID-vkCmdSetEvent-stageMask-04096
  If the task shaders feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

- VUID-vkCmdSetEvent-stageMask-04097
  If the shading rate image feature is not enabled, stageMask must not contain `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`

- VUID-vkCmdSetEvent-stageMask-4098
  Any pipeline stage included in stageMask must be supported by the capabilities of the queue family specified by the queueFamilyIndex member of the VkCommandPoolCreateInfo structure that was used to create the VkCommandPool that commandBuffer was allocated from, as specified in the table of supported pipeline stages.

- VUID-vkCmdSetEvent-stageMask-03937
  If the synchronization2 feature is not enabled, stageMask must not be 0

- VUID-vkCmdSetEvent-stageMask-01149
  stageMask must not include `VK_PIPELINE_STAGE_HOST_BIT`

- VUID-vkCmdSetEvent-commandBuffer-01152
  commandBuffer's current device mask must include exactly one physical device
Valid Usage (Implicit)

- VUID-vkCmdSetEvent-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetEvent-event-parameter
  `event` must be a valid `VkEvent` handle

- VUID-vkCmdSetEvent-stageMask-parameter
  `stageMask` must be a valid combination of `VkPipelineStageFlagBits` values

- VUID-vkCmdSetEvent-stageMask-requiredbitsetmask
  `stageMask` must not be 0

- VUID-vkCmdSetEvent-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetEvent-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- VUID-vkCmdSetEvent-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdSetEvent-commonparent
  Both of `commandBuffer`, and `event` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To unsignal the event from a device, call:
// Provided by VK_KHR_synchronization2
void vkCmdResetEvent2KHR(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags2KHR stageMask);

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be unsignaled.
- `stageMask` is a VkPipelineStageFlags2KHR mask of pipeline stages used to determine the first synchronization scope.

When `vkCmdResetEvent2KHR` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event unsignal operation which resets the event to the unsignaled state.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations by stage or stages that are logically earlier than stage.

The second synchronization scope includes only the event unsignal operation.

If `event` is already in the unsignaled state when `vkCmdResetEvent` is executed on the device, then `vkCmdResetEvent` has no effect, no event unsignal operation occurs, and no execution dependency is generated.
Valid Usage

- If the geometry shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`

- If the tessellation shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`

- If the conditional rendering feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`

- If the fragment density map feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

- If the transform feedback feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`

- If the mesh shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`

- If the task shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`

- If the shading rate image feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`

The synchronization2 feature must be enabled

- `stageMask` must not include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`

There must be an execution dependency between `vkCmdResetEvent2KHR` and the execution of any `vkCmdWaitEvents` that includes `event` in its `pEvents` parameter

There must be an execution dependency between `vkCmdResetEvent2KHR` and the execution of any `vkCmdWaitEvents2KHR` that includes `event` in its `pEvents` parameter

`commandBuffer`'s current device mask must include exactly one physical device.
Valid Usage (Implicit)

- VUID-vkCmdResetEvent2KHR-commandBuffer-parameter
  \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle

- VUID-vkCmdResetEvent2KHR-event-parameter
  \texttt{event} must be a valid \texttt{VkEvent} handle

- VUID-vkCmdResetEvent2KHR-stageMask-parameter
  \texttt{stageMask} must be a valid combination of \texttt{VkPipelineStageFlagBits2KHR} values

- VUID-vkCmdResetEvent2KHR-stageMask-requiredbitmask
  \texttt{stageMask} must not be 0

- VUID-vkCmdResetEvent2KHR-commandBuffer-recording
  \texttt{commandBuffer} must be in the recording state

- VUID-vkCmdResetEvent2KHR-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics, or compute operations

- VUID-vkCmdResetEvent2KHR-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdResetEvent2KHR-commonparent
  Both of \texttt{commandBuffer}, and \texttt{event} must have been created, allocated, or retrieved from the same \texttt{VkDevice}

Host Synchronization

- Host access to \texttt{commandBuffer} must be externally synchronized

- Host access to the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To set the state of an event to unsignaled from a device, call:
// Provided by VK_VERSION_1_0

```c
void vkCmdResetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `event` is the event that will be unsignaled.
- `stageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the source stage mask used to determine when the `event` is unsignaled.

`vkCmdResetEvent` behaves identically to `vkCmdResetEvent2KHR`. 

Chapter 7. Synchronization and Cache Control | 311
Valid Usage

- VUID-vkCmdResetEvent-stageMask-04090
  If the **geometry shaders** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- VUID-vkCmdResetEvent-stageMask-04091
  If the **tessellation shaders** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

- VUID-vkCmdResetEvent-stageMask-04092
  If the **conditional rendering** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`

- VUID-vkCmdResetEvent-stageMask-04093
  If the **fragment density map** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

- VUID-vkCmdResetEvent-stageMask-04094
  If the **transform feedback** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`

- VUID-vkCmdResetEvent-stageMask-04095
  If the **mesh shaders** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`

- VUID-vkCmdResetEvent-stageMask-04096
  If the **task shaders** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

- VUID-vkCmdResetEvent-stageMask-04097
  If the **shading rate image** feature is not enabled, `stageMask` **must** not contain `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`

- VUID-vkCmdResetEvent-stageMask-4098
  Any pipeline stage included in `stageMask` **must** be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the table of supported pipeline stages

- VUID-vkCmdResetEvent-stageMask-03937
  If the **synchronization2** feature is not enabled, `stageMask` **must** not be 0

- VUID-vkCmdResetEvent-stageMask-01153
  `stageMask` **must** not include `VK_PIPELINE_STAGE_HOST_BIT`

- VUID-vkCmdResetEvent-event-03834
  There **must** be an execution dependency between `vkCmdResetEvent` and the execution of any `vkCmdWaitEvents` that includes `event` in its `pEvents` parameter

- VUID-vkCmdResetEvent-event-03835
  There **must** be an execution dependency between `vkCmdResetEvent` and the execution of any `vkCmdWaitEvents2KHR` that includes `event` in its `pEvents` parameter

- VUID-vkCmdResetEvent-commandBuffer-01157
  `commandBuffer`’s current device mask **must** include exactly one physical device
Valid Usage (Implicit)

- **VUID-vkCmdResetEvent-commandBuffer-parameter**
  \textit{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle

- **VUID-vkCmdResetEvent-event-parameter**
  \textit{event} must be a valid \texttt{VkEvent} handle

- **VUID-vkCmdResetEvent-stageMask-parameter**
  \textit{stageMask} must be a valid combination of \texttt{VkPipelineStageFlagBits} values

- **VUID-vkCmdResetEvent-stageMask-requiredbitmask**
  \textit{stageMask} must not be 0

- **VUID-vkCmdResetEvent-commandBuffer-recording**
  \textit{commandBuffer} must be in the recording state

- **VUID-vkCmdResetEvent-commandBuffer-cmdpool**
  The \texttt{VkCommandPool} that \textit{commandBuffer} was allocated from must support graphics, or compute operations

- **VUID-vkCmdResetEvent-renderpass**
  This command must only be called outside of a render pass instance

- **VUID-vkCmdResetEvent-commonparent**
  Both of \textit{commandBuffer}, and \textit{event} must have been created, allocated, or retrieved from the same \texttt{VkDevice}

Host Synchronization

- Host access to \textit{commandBuffer} must be externally synchronized

- Host access to the \texttt{VkCommandPool} that \textit{commandBuffer} was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To wait for one or more events to enter the signaled state on a device, call:
// Provided by VK_KHR_synchronization2
void vkCmdWaitEvents2KHR(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    eventCount,
    const VkEvent*                              pEvents,
    const VkDependencyInfoKHR*                  pDependencyInfos);

• commandBuffer is the command buffer into which the command is recorded.
• eventCount is the length of the pEvents array.
• pEvents is an array of events to wait on.
• pDependencyInfos is an array of VkDependencyInfoKHR structures, defining the second synchronization scope.

When vkCmdWaitEvents2KHR is submitted to a queue, it inserts memory dependencies according to the elements of pDependencyInfos and each corresponding element of pEvents. vkCmdWaitEvents2KHR must not be used to wait on event signal operations occurring on other queues, or signal operations executed by vkCmdSetEvent.

The first synchronization scope and access scope of each memory dependency defined by any element i of pDependencyInfos are applied to operations commands that occurred earlier in submission order than the last event signal operation on element i of pEvents.

Signal operations for an event at index i are only included if:

• The event was signaled by a vkCmdSetEvent2KHR command that occurred earlier in submission order with a dependencyInfo parameter exactly equal to the element of pDependencyInfos at index i; or
• The event was created without VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR, and the first synchronization scope defined by the element of pDependencyInfos at index i only includes host operations (VK_PIPELINE_STAGE_2_HOST_BIT_KHR).

The second synchronization scope and access scope of each memory dependency defined by any element i of pDependencyInfos are applied to operations commands that occurred later in submission order than vkCmdWaitEvents2KHR.

Note

vkCmdWaitEvents2KHR is used with vkCmdSetEvent2KHR to define a memory dependency between two sets of action commands, roughly in the same way as pipeline barriers, but split into two commands such that work between the two may execute unhindered.
Note

Applications should be careful to avoid race conditions when using events. There is no direct ordering guarantee between `vkCmdSetEvent2KHR` and `vkCmdResetEvent2KHR`, `vkCmdResetEvent`, or `vkCmdSetEvent`. Another execution dependency (e.g. a pipeline barrier or semaphore with `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`) is needed to prevent such a race condition.
Valid Usage

- **VUID-vkCmdWaitEvents2KHR-synchronization2-03836**
  The synchronization2 feature must be enabled

- **VUID-vkCmdWaitEvents2KHR-pEvents-03837**
  Members of pEvents must not have been signaled by vkCmdSetEvent

- **VUID-vkCmdWaitEvents2KHR-pEvents-03838**
  For any element i of pEvents, if that event is signaled by vkCmdSetEvent2KHR, that command's dependencyInfo parameter must be exactly equal to the ith element of pDependencyInfos

- **VUID-vkCmdWaitEvents2KHR-pEvents-03839**
  For any element i of pEvents, if that event is signaled by vkSetEvent, barriers in the ith element of pDependencyInfos must include only host operations in their first synchronization scope

- **VUID-vkCmdWaitEvents2KHR-pEvents-03840**
  For any element i of pEvents, if barriers in the ith element of pDependencyInfos include only host operations, the ith element of pEvents must be signaled before vkCmdWaitEvents2KHR is executed

- **VUID-vkCmdWaitEvents2KHR-pEvents-03841**
  For any element i of pEvents, if barriers in the ith element of pDependencyInfos do not include host operations, the ith element of pEvents must be by a corresponding vkCmdSetEvent2KHR that occurred earlier in submission order

- **VUID-vkCmdWaitEvents2KHR-srcStageMask-03842**
  The srcStageMask member of any element of the pMemoryBarriers, pBufferMemoryBarriers, or pImageMemoryBarriers members of pDependencyInfo must either include only pipeline stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from, or include only VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- **VUID-vkCmdWaitEvents2KHR-dstStageMask-03843**
  The dstStageMask member of any element of the pMemoryBarriers, pBufferMemoryBarriers, or pImageMemoryBarriers members of pDependencyInfo must only include pipeline stages valid for the queue family that was used to create the command pool that commandBuffer was allocated from

- **VUID-vkCmdWaitEvents2KHR-dependencyFlags-03844**
  The dependencyFlags member of any element of dependencyInfo must be 0

- **VUID-vkCmdWaitEvents2KHR-pEvents-03845**
  If pEvents includes one or more events that will be signaled by vkSetEvent after commandBuffer has been submitted to a queue, then vkCmdWaitEvents2KHR must not be called inside a render pass instance

- **VUID-vkCmdWaitEvents2KHR-commandBuffer-03846**
  commandBuffer’s current device mask must include exactly one physical device
Valid Usage (Implicit)

- **VUID-vkCmdWaitEvents2KHR-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdWaitEvents2KHR-pEvents-parameter**
  `pEvents` must be a valid pointer to an array of `eventCount` valid `VkEvent` handles

- **VUID-vkCmdWaitEvents2KHR-pDependencyInfos-parameter**
  `pDependencyInfos` must be a valid pointer to an array of `eventCount` valid `VkDependencyInfoKHR` structures

- **VUID-vkCmdWaitEvents2KHR-commandBuffer-recording**
  `commandBuffer` must be in the recording state

- **VUID-vkCmdWaitEvents2KHR-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- **VUID-vkCmdWaitEvents2KHR-eventCount-arraylength**
  `eventCount` must be greater than 0

- **VUID-vkCmdWaitEvents2KHR-commonparent**
  Both of `commandBuffer`, and the elements of `pEvents` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To wait for one or more events to enter the signaled state on a device, call:
// Provided by VK_VERSION_1_0

void vkCmdWaitEvents(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    eventCount,
    constVkEvent*                              pEvents,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    uint32_t                                    memoryBarrierCount,
    constVkMemoryBarrier*                      pMemoryBarriers,
    uint32_t                                    bufferMemoryBarrierCount,
    constVkBufferMemoryBarrier*                pBufferMemoryBarriers,
    uint32_t                                    imageMemoryBarrierCount,
    constVkImageMemoryBarrier*                 pImageMemoryBarriers);

• commandBuffer is the command buffer into which the command is recorded.
• eventCount is the length of the pEvents array.
• pEvents is a pointer to an array of event object handles to wait on.
• srcStageMask is a bitmask of VkPipelineStageFlagBits specifying the source stage mask.
• dstStageMask is a bitmask of VkPipelineStageFlagBits specifying the destination stage mask.
• memoryBarrierCount is the length of the pMemoryBarriers array.
• pMemoryBarriers is a pointer to an array of VkMemoryBarrier structures.
• bufferMemoryBarrierCount is the length of the pBufferMemoryBarriers array.
• pBufferMemoryBarriers is a pointer to an array of VkBufferMemoryBarrier structures.
• imageMemoryBarrierCount is the length of the pImageMemoryBarriers array.
• pImageMemoryBarriers is a pointer to an array of VkImageMemoryBarrier structures.

vkCmdWaitEvents is largely similar to vkCmdWaitEvents2KHR, but can only wait on signal operations defined by vkCmdSetEvent. As vkCmdSetEvent doesn’t define any access scopes, vkCmdWaitEvents defines the first access scope for each event signal operation in addition to its own access scopes.

**Note**
Since vkCmdSetEvent doesn’t have any dependency information beyond a stage mask, implementations do not have the same opportunity to perform availability and visibility operations or image layout transitions in advance as they do with vkCmdSetEvent2KHR and vkCmdWaitEvents2KHR.

When vkCmdWaitEvents is submitted to a queue, it defines a memory dependency between prior event signal operations on the same queue or the host, and subsequent commands. vkCmdWaitEvents must not be used to wait on event signal operations occurring on other queues.

The first synchronization scope only includes event signal operations that operate on members of pEvents, and the operations that happened-before the event signal operations. Event signal operations performed by vkCmdSetEvent that occur earlier in submission order are included in the first synchronization scope, if the logically latest pipeline stage in their stageMask parameter is
logically earlier than or equal to the logically latest pipeline stage in \texttt{srcStageMask}. Event signal operations performed by \texttt{vkSetEvent} are only included in the first synchronization scope if \texttt{VK_PIPELINE_STAGE_HOST_BIT} is included in \texttt{srcStageMask}.

The second synchronization scope includes all commands that occur later in submission order. The second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by \texttt{dstStageMask}.

The first access scope is limited to access in the pipeline stages determined by the source stage mask specified by \texttt{srcStageMask}. Within that, the first access scope only includes the first access scopes defined by elements of the \texttt{pMemoryBarriers}, \texttt{pBufferMemoryBarriers} and \texttt{pImageMemoryBarriers} arrays, which each define a set of memory barriers. If no memory barriers are specified, then the first access scope includes no accesses.

The second access scope is limited to access in the pipeline stages determined by the destination stage mask specified by \texttt{dstStageMask}. Within that, the second access scope only includes the second access scopes defined by elements of the \texttt{pMemoryBarriers}, \texttt{pBufferMemoryBarriers} and \texttt{pImageMemoryBarriers} arrays, which each define a set of memory barriers. If no memory barriers are specified, then the second access scope includes no accesses.
Valid Usage

- **VUID-vkCmdWaitEvents-srcStageMask-04090**
  If the geometry shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

- **VUID-vkCmdWaitEvents-srcStageMask-04091**
  If the tessellation shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-vkCmdWaitEvents-srcStageMask-04092**
  If the conditional rendering feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`.

- **VUID-vkCmdWaitEvents-srcStageMask-04093**
  If the fragment density map feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`.

- **VUID-vkCmdWaitEvents-srcStageMask-04094**
  If the transform feedback feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`.

- **VUID-vkCmdWaitEvents-srcStageMask-04095**
  If the mesh shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`.

- **VUID-vkCmdWaitEvents-srcStageMask-04096**
  If the task shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`.

- **VUID-vkCmdWaitEvents-srcStageMask-04097**
  If the shading rate image feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`.

- **VUID-vkCmdWaitEvents-srcStageMask-04098**
  Any pipeline stage included in `srcStageMask` must be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the table of supported pipeline stages.

- **VUID-vkCmdWaitEvents-srcStageMask-03937**
  If the synchronization2 feature is not enabled, `srcStageMask` must not be 0.

- **VUID-vkCmdWaitEvents-dstStageMask-04090**
  If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

- **VUID-vkCmdWaitEvents-dstStageMask-04091**
  If the tessellation shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-vkCmdWaitEvents-dstStageMask-04092**
  If the conditional rendering feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`.

- **VUID-vkCmdWaitEvents-dstStageMask-04093**
  If the synchronization2 feature is not enabled, `dstStageMask` must not be 0.
If the fragment density map feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT

- VUID-vkCmdWaitEvents-dstStageMask-04094

If the transform feedback feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT

- VUID-vkCmdWaitEvents-dstStageMask-04095

If the mesh shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV

- VUID-vkCmdWaitEvents-dstStageMask-04096

If the task shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV

- VUID-vkCmdWaitEvents-dstStageMask-04097

If the shading rate image feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV

- VUID-vkCmdWaitEvents-dstStageMask-4098

Any pipeline stage included in dstStageMask must be supported by the capabilities of the queue family specified by the queueFamilyIndex member of the VkCommandPoolCreateInfo structure that was used to create the VkCommandPool that commandBuffer was allocated from, as specified in the table of supported pipeline stages

- VUID-vkCmdWaitEvents-dstStageMask-03937

If the synchronization2 feature is not enabled, dstStageMask must not be 0

- VUID-vkCmdWaitEvents-srcAccessMask-02815

The srcAccessMask member of each element of pMemoryBarriers must only include access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types

- VUID-vkCmdWaitEvents-dstAccessMask-02816

The dstAccessMask member of each element of pMemoryBarriers must only include access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types

- VUID-vkCmdWaitEvents-pBufferMemoryBarriers-02817

For any element of pBufferMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its srcQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its srcAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in srcStageMask, as specified in the table of supported access types

- VUID-vkCmdWaitEvents-pBufferMemoryBarriers-02818

For any element of pBufferMemoryBarriers, if its srcQueueFamilyIndex and dstQueueFamilyIndex members are equal, or if its dstQueueFamilyIndex is the queue family index that was used to create the command pool that commandBuffer was allocated from, then its dstAccessMask member must only contain access flags that are supported by one or more of the pipeline stages in dstStageMask, as specified in the table of supported access types

- VUID-vkCmdWaitEvents-pImageMemoryBarriers-02819
For any element of `pImageMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `srcQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `srcAccessMask` member must only contain access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the table of supported access types.

- VUID-vkCmdWaitEvents-pImageMemoryBarriers-02820

For any element of `pImageMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `dstQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `dstAccessMask` member must only contain access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the table of supported access types.

- VUID-vkCmdWaitEvents-srcStageMask-01158
  `srcStageMask` must be the bitwise OR of the `stageMask` parameter used in previous calls to `vkCmdSetEvent` with any of the members of `pEvents` and `VK_PIPELINE_STAGE_HOST_BIT` if any of the members of `pEvents` was set using `vkSetEvent`.

- VUID-vkCmdWaitEvents-pEvents-01163
  If `pEvents` includes one or more events that will be signaled by `vkSetEvent` after `commandBuffer` has been submitted to a queue, then `vkCmdWaitEvents` must not be called inside a render pass instance.

- VUID-vkCmdWaitEvents-srcQueueFamilyIndex-02803
  The `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any element of `pBufferMemoryBarriers` or `pImageMemoryBarriers` must be equal.

- VUID-vkCmdWaitEvents-commandBuffer-01167
  `commandBuffer`’s current device mask must include exactly one physical device.

- VUID-vkCmdWaitEvents-pEvents-03847
  Members of `pEvents` must not have been signaled by `vkCmdSetEvent2KHR`.

322 | Chapter 7. Synchronization and Cache Control
Valid Usage (Implicit)

- **VUID-vkCmdWaitEvents-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdWaitEvents-pEvents-parameter**
  - `pEvents` must be a valid pointer to an array of `eventCount` valid `VkEvent` handles

- **VUID-vkCmdWaitEvents-srcStageMask-parameter**
  - `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits` values

- **VUID-vkCmdWaitEvents-dstStageMask-parameter**
  - `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits` values

- **VUID-vkCmdWaitEvents-pMemoryBarriers-parameter**
  - If `memoryBarrierCount` is not 0, `pMemoryBarriers` must be a valid pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures

- **VUID-vkCmdWaitEvents-pBufferMemoryBarriers-parameter**
  - If `bufferMemoryBarrierCount` is not 0, `pBufferMemoryBarriers` must be a valid pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures

- **VUID-vkCmdWaitEvents-pImageMemoryBarriers-parameter**
  - If `imageMemoryBarrierCount` is not 0, `pImageMemoryBarriers` must be a valid pointer to an array of `imageMemoryBarrierCount` valid `VkImageMemoryBarrier` structures

- **VUID-vkCmdWaitEvents-commandBuffer-recording**
  - `commandBuffer` must be in the recording state

- **VUID-vkCmdWaitEvents-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- **VUID-vkCmdWaitEvents-eventCount-arraylength**
  - `eventCount` must be greater than 0

- **VUID-vkCmdWaitEvents-commonparent**
  - Both of `commandBuffer`, and the elements of `pEvents` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
7.6. Pipeline Barriers

To record a pipeline barrier, call:

```c
// Provided by VK_KHR_synchronization2
void vkCmdPipelineBarrier2KHR(
    VkCommandBuffer commandBuffer,
    const VkDependencyInfoKHR* pDependencyInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pDependencyInfo` is a pointer to a `VkDependencyInfoKHR` structure defining the scopes of this operation.

When `vkCmdPipelineBarrier2KHR` is submitted to a queue, it defines memory dependencies between commands that were submitted before it, and those submitted after it.

The first **synchronization scope** and **access scope** of each memory dependency defined by any element `i` of `pDependencyInfos` are applied to operations commands that occurred earlier in submission order.

The second **synchronization scope** and **access scope** of each memory dependency defined by any element `i` of `pDependencyInfos` are applied to operations commands that occurred later in submission order.

If `vkCmdPipelineBarrier2KHR` is recorded within a render pass instance, the synchronization scopes are **limited to operations within the same subpass**.
Valid Usage

- **VUID-vkCmdPipelineBarrier2KHR-pDependencies-02285**
  If *vkCmdPipelineBarrier2KHR* is called within a render pass instance, the render pass **must** have been created with at least one *VkSubpassDependency* instance in *VkRenderPassCreateInfo::pDependencies* that expresses a dependency from the current subpass to itself, with synchronization scopes and access scopes that are all supersets of the scopes defined in this command.

- **VUID-vkCmdPipelineBarrier2KHR-bufferMemoryBarrierCount-01178**
  If *vkCmdPipelineBarrier2KHR* is called within a render pass instance, it **must** not include any buffer memory barriers.

- **VUID-vkCmdPipelineBarrier2KHR-image-04073**
  If *vkCmdPipelineBarrier2KHR* is called within a render pass instance, the *image* member of any image memory barrier included in this command **must** be an attachment used in the current subpass both as an input attachment, and as either a color or depth/stencil attachment.

- **VUID-vkCmdPipelineBarrier2KHR-oldLayout-01181**
  If *vkCmdPipelineBarrier2KHR* is called within a render pass instance, the *oldLayout* and *newLayout* members of any image memory barrier included in this command **must** be equal.

- **VUID-vkCmdPipelineBarrier2KHR-srcQueueFamilyIndex-01182**
  If *vkCmdPipelineBarrier2KHR* is called within a render pass instance, the *srcQueueFamilyIndex* and *dstQueueFamilyIndex* members of any image memory barrier included in this command **must** be equal.

- **VUID-vkCmdPipelineBarrier2KHR-dependencyFlags-01186**
  If *vkCmdPipelineBarrier2KHR* is called outside of a render pass instance, **VK_DEPENDENCY_VIEW_LOCAL_BIT** must not be included in the dependency flags.

- **VUID-vkCmdPipelineBarrier2KHR-synchronization2-03848**
  The *synchronization2* feature **must** be enabled.

- **VUID-vkCmdPipelineBarrier2KHR-srcStageMask-03849**
  The *srcStageMask* member of any element of the *pMemoryBarriers*, *pBufferMemoryBarriers*, *pImageMemoryBarriers* members of *pDependencyInfo* **must** only include pipeline stages valid for the queue family that was used to create the command pool that *commandBuffer* was allocated from.

- **VUID-vkCmdPipelineBarrier2KHR-dstStageMask-03850**
  The *dstStageMask* member of any element of the *pMemoryBarriers*, *pBufferMemoryBarriers*, *pImageMemoryBarriers* members of *pDependencyInfo* **must** only include pipeline stages valid for the queue family that was used to create the command pool that *commandBuffer* was allocated from.
Valid Usage (Implicit)

- **VUID-vkCmdPipelineBarrier2KHR-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdPipelineBarrier2KHR-pDependencyInfo-parameter**
  
  `pDependencyInfo` must be a valid pointer to a valid `VkDependencyInfoKHR` structure.

- **VUID-vkCmdPipelineBarrier2KHR-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdPipelineBarrier2KHR-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations.

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer, Graphics, Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record a pipeline barrier, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdPipelineBarrier(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    VkDependencyFlags dependencyFlags,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```

- `commandBuffer` is the command buffer into which the command is recorded.
• **srcStageMask** is a bitmask of VkPipelineStageFlagBits specifying the *source stages*.

• **dstStageMask** is a bitmask of VkPipelineStageFlagBits specifying the *destination stages*.

• **dependencyFlags** is a bitmask of VkDependencyFlagBits specifying how execution and memory dependencies are formed.

• **memoryBarrierCount** is the length of the pMemoryBarriers array.

• **pMemoryBarriers** is a pointer to an array of VkMemoryBarrier structures.

• **bufferMemoryBarrierCount** is the length of the pBufferMemoryBarriers array.

• **pBufferMemoryBarriers** is a pointer to an array of VkBufferMemoryBarrier structures.

• **imageMemoryBarrierCount** is the length of the pImageMemoryBarriers array.

• **pImageMemoryBarriers** is a pointer to an array of VkImageMemoryBarrier structures.

**vkCmdPipelineBarrier** operates almost identically to **vkCmdPipelineBarrier2KHR**, other than the scopes and barriers defined as direct parameters rather than being defined by an VkDependencyInfoKHR.

When **vkCmdPipelineBarrier** is submitted to a queue, it defines a memory dependency between commands that were submitted before it, and those submitted after it.

If **vkCmdPipelineBarrier** was recorded outside a render pass instance, the first *synchronization scope* includes all commands that occur earlier in submission order. If **vkCmdPipelineBarrier** was recorded inside a render pass instance, the first synchronization scope includes only commands that occur earlier in submission order within the same subpass. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the *source stage mask* specified by **srcStageMask**.

If **vkCmdPipelineBarrier** was recorded outside a render pass instance, the second *synchronization scope* includes all commands that occur later in submission order. If **vkCmdPipelineBarrier** was recorded inside a render pass instance, the second synchronization scope includes only commands that occur later in submission order within the same subpass. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the *destination stage mask* specified by **dstStageMask**.

The first *access scope* is limited to access in the pipeline stages determined by the *source stage mask* specified by **srcStageMask**. Within that, the first access scope only includes the first access scopes defined by elements of the pMemoryBarriers, pBufferMemoryBarriers and pImageMemoryBarriers arrays, which each define a set of *memory barriers*. If no memory barriers are specified, then the first access scope includes no accesses.

The second *access scope* is limited to access in the pipeline stages determined by the *destination stage mask* specified by **dstStageMask**. Within that, the second access scope only includes the second access scopes defined by elements of the pMemoryBarriers, pBufferMemoryBarriers and pImageMemoryBarriers arrays, which each define a set of *memory barriers*. If no memory barriers are specified, then the second access scope includes no accesses.

If **dependencyFlags** includes VK_DEPENDENCY_BY_REGION_BIT, then any dependency between framebuffer-space pipeline stages is framebuffer-local - otherwise it is framebuffer-global.
Valid Usage

• **VUID-vkCmdPipelineBarrier-srcStageMask-04090**
  
  If the **geometry shaders** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04091**
  
  If the **tessellation shaders** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04092**
  
  If the **conditional rendering** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04093**
  
  If the **fragment density map** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04094**
  
  If the **transform feedback** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04095**
  
  If the **mesh shaders** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04096**
  
  If the **task shaders** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04097**
  
  If the **shading rate image** feature is not enabled, `srcStageMask` **must** not contain `VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV`

• **VUID-vkCmdPipelineBarrier-srcStageMask-04098**
  
  Any pipeline stage included in `srcStageMask` **must** be supported by the capabilities of the queue family specified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` structure that was used to create the `VkCommandPool` that `commandBuffer` was allocated from, as specified in the **table of supported pipeline stages**

• **VUID-vkCmdPipelineBarrier-srcStageMask-03937**
  
  If the **synchronization2** feature is not enabled, `srcStageMask` **must** not be `0`

• **VUID-vkCmdPipelineBarrier-dstStageMask-04090**
  
  If the **geometry shaders** feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

• **VUID-vkCmdPipelineBarrier-dstStageMask-04091**
  
  If the **tessellation shaders** feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`

• **VUID-vkCmdPipelineBarrier-dstStageMask-04092**
  
  If the **conditional rendering** feature is not enabled, `dstStageMask` **must** not contain `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`

• **VUID-vkCmdPipelineBarrier-dstStageMask-04093**
  
  ...
If the fragment density map feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT}

- VUID-vkCmdPipelineBarrier-dstStageMask-04094

If the transform feedback feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT}

- VUID-vkCmdPipelineBarrier-dstStageMask-04095

If the mesh shaders feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV}

- VUID-vkCmdPipelineBarrier-dstStageMask-04096

If the task shaders feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV}

- VUID-vkCmdPipelineBarrier-dstStageMask-04097

If the shading rate image feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV}

- VUID-vkCmdPipelineBarrier-dstStageMask-4098

Any pipeline stage included in \texttt{dstStageMask} must be supported by the capabilities of the queue family specified by the \texttt{queueFamilyIndex} member of the \texttt{VkCommandPoolCreateInfo} structure that was used to create the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from, as specified in the table of supported pipeline stages

- VUID-vkCmdPipelineBarrier-dstStageMask-03937

If the synchronization2 feature is not enabled, \texttt{dstStageMask} must not be 0

- VUID-vkCmdPipelineBarrier-srcAccessMask-02815

The \texttt{srcAccessMask} member of each element of \texttt{pMemoryBarriers} must only include access flags that are supported by one or more of the pipeline stages in \texttt{srcStageMask}, as specified in the table of supported access types

- VUID-vkCmdPipelineBarrier-dstAccessMask-02816

The \texttt{dstAccessMask} member of each element of \texttt{pMemoryBarriers} must only include access flags that are supported by one or more of the pipeline stages in \texttt{dstStageMask}, as specified in the table of supported access types

- VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-02817

For any element of \texttt{pBufferMemoryBarriers}, if its \texttt{srcQueueFamilyIndex} and \texttt{dstQueueFamilyIndex} members are equal, or if its \texttt{srcQueueFamilyIndex} is the queue family index that was used to create the command pool that \texttt{commandBuffer} was allocated from, then its \texttt{srcAccessMask} member must only contain access flags that are supported by one or more of the pipeline stages in \texttt{srcStageMask}, as specified in the table of supported access types

- VUID-vkCmdPipelineBarrier-dstQueueFamilyIndex-02818

For any element of \texttt{pBufferMemoryBarriers}, if its \texttt{srcQueueFamilyIndex} and \texttt{dstQueueFamilyIndex} members are equal, or if its \texttt{dstQueueFamilyIndex} is the queue family index that was used to create the command pool that \texttt{commandBuffer} was allocated from, then its \texttt{dstAccessMask} member must only contain access flags that are supported by one or more of the pipeline stages in \texttt{dstStageMask}, as specified in the table of supported access types

- VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-02819
For any element of `pImageMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `srcQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `srcAccessMask` member must only contain access flags that are supported by one or more of the pipeline stages in `srcStageMask`, as specified in the table of supported access types

- [VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-02820](#)

For any element of `pImageMemoryBarriers`, if its `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members are equal, or if its `dstQueueFamilyIndex` is the queue family index that was used to create the command pool that `commandBuffer` was allocated from, then its `dstAccessMask` member must only contain access flags that are supported by one or more of the pipeline stages in `dstStageMask`, as specified in the table of supported access types

- [VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-02820](#)

If `vkCmdPipelineBarrier` is called within a render pass instance, the render pass must have been created with at least one `VkSubpassDependency` instance in `VkRenderPassCreateInfo::pDependencies` that expresses a dependency from the current subpass to itself, with synchronization scopes and access scopes that are all supersets of the scopes defined in this command

- [VUID-vkCmdPipelineBarrier-pDependencies-02285](#)

If `vkCmdPipelineBarrier` is called within a render pass instance, it must not include any buffer memory barriers

- [VUID-vkCmdPipelineBarrier-bufferMemoryBarrierCount-01178](#)

If `vkCmdPipelineBarrier` is called within a render pass instance, the `image` member of any image memory barrier included in this command must be an attachment used in the current subpass both as an input attachment, and as either a color or depth/stencil attachment

- [VUID-vkCmdPipelineBarrier-image-04073](#)

If `vkCmdPipelineBarrier` is called within a render pass instance, the `oldLayout` and `newLayout` members of any image memory barrier included in this command must be equal

- [VUID-vkCmdPipelineBarrier-oldLayout-01181](#)

If `vkCmdPipelineBarrier` is called within a render pass instance, the `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any image memory barrier included in this command must be equal

- [VUID-vkCmdPipelineBarrier-srcQueueFamilyIndex-01182](#)

If `vkCmdPipelineBarrier` is called outside of a render pass instance, `VK_DEPENDENCY_VIEW_LOCAL_BIT` must not be included in the dependency flags

- [VUID-vkCmdPipelineBarrier-dependencyFlags-01186](#)
Valid Usage (Implicit)

- VUID-vkCmdPipelineBarrier-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdPipelineBarrier-srcStageMask-parameter
  srcStageMask must be a valid combination of VkPipelineStageFlagBits values

- VUID-vkCmdPipelineBarrier-srcStageMask-requiredbitmask
  srcStageMask must not be 0

- VUID-vkCmdPipelineBarrier-dstStageMask-parameter
  dstStageMask must be a valid combination of VkPipelineStageFlagBits values

- VUID-vkCmdPipelineBarrier-dstStageMask-requiredbitmask
  dstStageMask must not be 0

- VUID-vkCmdPipelineBarrier-dependencyFlags-parameter
  dependencyFlags must be a valid combination of VkDependencyFlagBits values

- VUID-vkCmdPipelineBarrier-pMemoryBarriers-parameter
  If memoryBarrierCount is not 0, pMemoryBarriers must be a valid pointer to an array of memoryBarrierCount valid VkMemoryBarrier structures

- VUID-vkCmdPipelineBarrier-pBufferMemoryBarriers-parameter
  If bufferMemoryBarrierCount is not 0, pBufferMemoryBarriers must be a valid pointer to an array of bufferMemoryBarrierCount valid VkBufferMemoryBarrier structures

- VUID-vkCmdPipelineBarrier-pImageMemoryBarriers-parameter
  If imageMemoryBarrierCount is not 0, pImageMemoryBarriers must be a valid pointer to an array of imageMemoryBarrierCount valid VkImageMemoryBarrier structures

- VUID-vkCmdPipelineBarrier-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdPipelineBarrier-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
Bits which can be set in `vkCmdPipelineBarrier::dependencyFlags`, specifying how execution and memory dependencies are formed, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDependencyFlagBits {
    VKDEPENDENCYBYREGIONBIT = 0x00000001,
    // Provided by VK_VERSION_1_1
    VKDEPENDENCYDEVICEGROUPBIT = 0x00000004,
    // Provided by VK_VERSION_1_1
    VKDEPENDENCYVIEWLOCALBIT = 0x00000002,
    // Provided by VK_KHR_multiview
    VKDEPENDENCYVIEWLOCALBITKHR = VKDEPENDENCYVIEWLOCALBIT,
    // Provided by VK_KHR_device_group
    VKDEPENDENCYDEVICEGROUPBIT_KHR = VKDEPENDENCYDEVICEGROUPBIT,
} VkDependencyFlagBits;
```

- **VK_DEPENDENCY_BY_REGION_BIT** specifies that dependencies will be framebuffer-local.
- **VK_DEPENDENCY_VIEW_LOCAL_BIT** specifies that a subpass has more than one view.
- **VK_DEPENDENCY_DEVICE_GROUP_BIT** specifies that dependencies are non-device-local dependency.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkDependencyFlags;
```

`VkDependencyFlags` is a bitmask type for setting a mask of zero or more `VkDependencyFlagBits`.

### 7.6.1. Subpass Self-dependency

If `vkCmdPipelineBarrier` or `vkCmdPipelineBarrier2KHR` is called inside a render pass instance, the following restrictions apply. For a given subpass to allow a pipeline barrier, the render pass must declare a self-dependency from that subpass to itself. That is, there must exist a subpass dependency with `srcSubpass` and `dstSubpass` both equal to that subpass index. More than one self-dependency can be declared for each subpass.

Self-dependencies must only include pipeline stage bits that are graphics stages. If any of the stages in `srcStages` are framebuffer-space stages, `dstStages` must only contain framebuffer-space stages. This means that pseudo-stages like `VK_PIPELINE_STAGE_ALL_COMMANDS_BIT` which include the execution...
of both framebuffer-space stages and non-framebuffer-space stages must not be used.

If the source and destination stage masks both include framebuffer-space stages, then dependencyFlags must include VK_DEPENDENCY_BY_REGION_BIT. If the subpass has more than one view, then dependencyFlags must include VK_DEPENDENCY_VIEW_LOCAL_BIT.

Each of the synchronization scopes and access scopes of a vkCmdPipelineBarrier2KHR or vkCmdPipelineBarrier command inside a render pass instance must be a subset of the scopes of one of the self-dependencies for the current subpass.

If the self-dependency has VK_DEPENDENCY_BY_REGION_BIT or VK_DEPENDENCY_VIEW_LOCAL_BIT set, then so must the pipeline barrier. Pipeline barriers within a render pass instance must not include buffer memory barriers. Image memory barriers must only specify image subresources that are used as attachments within the subpass, and must not define an image layout transition or queue family ownership transfer.

### 7.7. Memory Barriers

**Memory barriers** are used to explicitly control access to buffer and image subresource ranges. Memory barriers are used to transfer ownership between queue families, change image layouts, and define availability and visibility operations. They explicitly define the access types and buffer and image subresource ranges that are included in the access scopes of a memory dependency that is created by a synchronization command that includes them.

#### 7.7.1. Global Memory Barriers

Global memory barriers apply to memory accesses involving all memory objects that exist at the time of its execution.

The VkMemoryBarrier2KHR structure is defined as:

```c
typedef struct VkMemoryBarrier2KHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkPipelineStageFlags2KHR    srcStageMask;
    VkAccessFlags2KHR           srcAccessMask;
    VkPipelineStageFlags2KHR    dstStageMask;
    VkAccessFlags2KHR           dstAccessMask;
} VkMemoryBarrier2KHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcStageMask** is a VkPipelineStageFlags2KHR mask of pipeline stages to be included in the first synchronization scope.
- **srcAccessMask** is a VkAccessFlags2KHR mask of pipeline first access scope.
- **dstStageMask** is a VkPipelineStageFlags2KHR mask of pipeline stages to be included in the image layout transition or queue family ownership transfer.
second synchronization scope.

- **dstStageMask** is a VkAccessFlags2KHR mask of pipeline second access scope.

This structure defines a memory dependency affecting all device memory.

The first synchronization scope and access scope described by this structure include only operations and memory accesses specified by **srcStageMask** and **srcAccessMask**.

The second synchronization scope and access scope described by this structure include only operations and memory accesses specified by **dstStageMask** and **dstAccessMask**.

---

**Valid Usage (Implicit)**

- **VUID-VkMemoryBarrier2KHR-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_MEMORY_BARRIER_2_KHR**

- **VUID-VkMemoryBarrier2KHR-srcStageMask-parameter**
  
  **srcStageMask** must be a valid combination of **VkPipelineStageFlagBits2KHR** values

- **VUID-VkMemoryBarrier2KHR-srcAccessMask-parameter**
  
  **srcAccessMask** must be a valid combination of **VkAccessFlagBits2KHR** values

- **VUID-VkMemoryBarrier2KHR-dstStageMask-parameter**
  
  **dstStageMask** must be a valid combination of **VkPipelineStageFlagBits2KHR** values

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-parameter**
  
  **dstAccessMask** must be a valid combination of **VkAccessFlagBits2KHR** values
Valid Usage

- If the **geometry shaders** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.
- If the **tessellation shaders** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.
- If the **conditional rendering** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`.
- If the **transform feedback** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`.
- If the **mesh shaders** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`.
- If the **task shaders** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`.
- If the **shading rate image** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`.
- If `srcAccessMask` includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
- If `srcAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
- If `srcAccessMask` includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
- If `srcAccessMask` includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
If `srcAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`. 
VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03914
  If srcAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03915
  If srcAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR,
  VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03916
  If srcAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03917
  If srcAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03918
  If srcAccessMask includes VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT, srcStageMask
  must include VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03919
  If srcAccessMask includes VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT, srcStageMask
  must include VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03920
  If srcAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT, srcStageMask
  must include VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-04747
  If srcAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT,
  srcStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR,
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03922
  If srcAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT,
  srcStageMask must include
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03923
  If srcAccessMask includes VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV, srcStageMask must
  include
  VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03924
  If `srcAccessMask` includes VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV, `srcStageMask` must include VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03925
  If `srcAccessMask` includes VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV, `srcStageMask` must include VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03926
  If `srcAccessMask` includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT, `srcStageMask` must include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03927
  If `srcAccessMask` includes VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR, `srcStageMask` must include VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-03928
  If `srcAccessMask` includes VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR, `srcStageMask` must include VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-04858
  If `srcAccessMask` includes VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR, `srcStageMask` must include VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-04859
  If `srcAccessMask` includes VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR, `srcStageMask` must include VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-04860
  If `srcAccessMask` includes VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR, `srcStageMask` must include VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

- VUID-VkMemoryBarrier2KHR-srcAccessMask-04861
  If `srcAccessMask` includes VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR, `srcStageMask` must include VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstStageMask-03929
  If the **geometry shaders** feature is not enabled, `dstStageMask` must not contain VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstStageMask-03930
  If the **tessellation shaders** feature is not enabled, `dstStageMask` must not contain VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstStageMask-03931
  If the **conditional rendering** feature is not enabled, `dstStageMask` must not contain VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT
If the fragment density map feature is not enabled, \( \text{dstStageMask} \) must not contain `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`.

If the transform feedback feature is not enabled, \( \text{dstStageMask} \) must not contain `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`.

If the mesh shaders feature is not enabled, \( \text{dstStageMask} \) must not contain `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`.

If the task shaders feature is not enabled, \( \text{dstStageMask} \) must not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`.

If the shading rate image feature is not enabled, \( \text{dstStageMask} \) must not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_UNIFORM_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.

If \( \text{dstAccessMask} \) includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR`, \( \text{dstStageMask} \) must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages.
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03907
  If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03908
  If dstAccessMask includes VK_ACCESS_2_SHADER_READ_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03909
  If dstAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03910
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03911
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03912
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03913
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03914
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkMemoryBarrier2KHR-dstAccessMask-03915
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, dstStageMask must include:
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR,
VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03916
If dstAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, dstStageMask must include
VK_PIPELINE_STAGE_2_HOST_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03917
If dstAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, dstStageMask must include
VK_PIPELINE_STAGE_2_HOST_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03918
If dstAccessMask includes VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT, dstStageMask
must include VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03919
If dstAccessMask includes VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT, dstStageMask
must include VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03920
If dstAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT, dstStageMask
must include VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-04747
If dstAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT, dstStageMask
must include VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR,
VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03922
If dstAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT, dstStageMask
must include VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03923
If dstAccessMask includes VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV, dstStageMask must
include VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03924
If dstAccessMask includes VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV, dstStageMask must
include VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV,
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03925
If dstAccessMask includes VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV, dstStageMask must
include VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV,
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkMemoryBarrier2KHR-dstAccessMask-03926
If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT, dstStageMask must
include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR
The `VkMemoryBarrier` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryBarrier {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
} VkMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.

The first access scope is limited to access types in the source access mask specified by `srcAccessMask`.

The second access scope is limited to access types in the destination access mask specified by `dstAccessMask`.

---

VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03927**
  
  If `dstAccessMask` includes `VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-03928**
  
  If `dstAccessMask` includes `VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-04858**
  
  If `dstAccessMask` includes `VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR`.

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-04859**
  
  If `dstAccessMask` includes `VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR`.

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-04860**
  
  If `dstAccessMask` includes `VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR`.

- **VUID-VkMemoryBarrier2KHR-dstAccessMask-04861**
  
  If `dstAccessMask` includes `VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR`.  

---

The `VkMemoryBarrier2KHR` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryBarrier2KHR {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
    VkAccessFlags      srcStageMask;
    VkAccessFlags      dstStageMask;
} VkMemoryBarrier2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
- `srcStageMask` is a bitmask of `VkAccessFlagBits` specifying a source stage mask.
- `dstStageMask` is a bitmask of `VkAccessFlagBits` specifying a destination stage mask.

The first access scope is limited to access types in the source access mask specified by `srcAccessMask`.

The second access scope is limited to access types in the destination access mask specified by `dstAccessMask`.

---

The `VkMemoryBarrier2KHR` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryBarrier2KHR {
    VkStructureType    sType;
    const void*        pNext;
    VkAccessFlags      srcAccessMask;
    VkAccessFlags      dstAccessMask;
    VkAccessFlags      srcStageMask;
    VkAccessFlags      dstStageMask;
} VkMemoryBarrier2KHR;
```
Valid Usage (Implicit)

- VUID-VkMemoryBarrier-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_MEMORY_BARRIER`

- VUID-VkMemoryBarrier-pNext-pNext
  
  `pNext` must be `NULL`

- VUID-VkMemoryBarrier-srcAccessMask-parameter
  
  `srcAccessMask` must be a valid combination of `VkAccessFlagBits` values

- VUID-VkMemoryBarrier-dstAccessMask-parameter
  
  `dstAccessMask` must be a valid combination of `VkAccessFlagBits` values

7.7.2. Buffer Memory Barriers

Buffer memory barriers only apply to memory accesses involving a specific buffer range. That is, a memory dependency formed from a buffer memory barrier is scoped to access via the specified buffer range. Buffer memory barriers can also be used to define a queue family ownership transfer for the specified buffer range.

The `VkBufferMemoryBarrier2KHR` structure is defined as:

```
// Provided by VK_KHR_synchronization2
typedef struct VkBufferMemoryBarrier2KHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkPipelineStageFlags2KHR    srcStageMask;
    VkAccessFlags2KHR           srcAccessMask;
    VkPipelineStageFlags2KHR    dstStageMask;
    VkAccessFlags2KHR           dstAccessMask;
    uint32_t                    srcQueueFamilyIndex;
    uint32_t                    dstQueueFamilyIndex;
    VkBuffer                    buffer;
    VkDeviceSize                offset;
    VkDeviceSize                size;
} VkBufferMemoryBarrier2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcStageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the first synchronization scope.
- `srcAccessMask` is a `VkAccessFlags2KHR` mask of pipeline first access scope.
- `dstStageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the second synchronization scope.
- `dstStageMask` is a `VkAccessFlags2KHR` mask of pipeline second access scope.
- `srcQueueFamilyIndex` is the source queue family for a *queue family ownership transfer*.
- `dstQueueFamilyIndex` is the destination queue family for a *queue family ownership transfer*.
- `buffer` is a handle to the buffer whose backing memory is affected by the barrier.
- `offset` is an offset in bytes into the backing memory for `buffer`; this is relative to the base offset as bound to the buffer (see `vkBindBufferMemory`).
- `size` is a size in bytes of the affected area of backing memory for `buffer`, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

This structure defines a *memory dependency* limited to a range of a buffer, and optionally defines a *queue family transfer operation* for that range.

The first *synchronization scope* and *access scope* described by this structure include only operations and memory accesses specified by elements `srcStageMask` and `srcAccessMask`.

The second *synchronization scope* and *access scope* described by this structure include only operations and memory accesses specified by elements `dstStageMask` and `dstAccessMask`.

Both *access scopes* are limited to only memory accesses to `buffer` in the range defined by `offset` and `size`.

If `buffer` was created with `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, this memory barrier defines a *queue family transfer operation*. When executed on a queue in the family identified by `srcQueueFamilyIndex`, this barrier defines a *queue family release operation* for the specified buffer range, and the second synchronization and access scopes do not synchronize operations on that queue. When executed on a queue in the family identified by `dstQueueFamilyIndex`, this barrier defines a *queue family acquire operation* for the specified buffer range, and the first synchronization and access scopes do not synchronize operations on that queue.

A *queue family transfer operation* is also defined if the values are not equal, and either is one of the special queue family values reserved for external memory ownership transfers, as described in *Queue Family Ownership Transfer*. A *queue family release operation* is defined when `dstQueueFamilyIndex` is one of those values, and a *queue family acquire operation* is defined when `srcQueueFamilyIndex` is one of those values.
Valid Usage

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03929
  If the geometry shaders feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03930
  If the tessellation shaders feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR
  or
  VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03931
  If the conditional rendering feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03932
  If the fragment density map feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03933
  If the transform feedback feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03934
  If the mesh shaders feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03935
  If the task shaders feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV

- VUID-VkBufferMemoryBarrier2KHR-srcStageMask-03936
  If the shading rate image feature is not enabled, srcStageMask must not contain
  VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV

- VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03900
  If srcAccessMask includes VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03901
  If srcAccessMask includes VK_ACCESS_2_INDEX_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03902
  If srcAccessMask includes VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03903
  If srcAccessMask includes VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR
If `srcAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_2_*_SHADER_BIT` stages.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03914**
  If `srcAccessMask` includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03915**
  If `srcAccessMask` includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03916**
  If `srcAccessMask` includes VK_ACCESS_2_HOST_READ_BIT_KHR, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03917**
  If `srcAccessMask` includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03918**
  If `srcAccessMask` includes VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03919**
  If `srcAccessMask` includes VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03920**
  If `srcAccessMask` includes VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-04747**
  If `srcAccessMask` includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR, VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03922**
  If `srcAccessMask` includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- **VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03923**
  If `srcAccessMask` includes VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV, `srcStageMask` must include
  VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV,
• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03924
  If srcAccessMask includes VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV, srcStageMask must include
  VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03925
  If srcAccessMask includes VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV, srcStageMask must include
  VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03926
  If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03927
  If srcAccessMask includes VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-03928
  If srcAccessMask includes VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-04858
  If srcAccessMask includes VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-04859
  If srcAccessMask includes VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-04860
  If srcAccessMask includes VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-srcAccessMask-04861
  If srcAccessMask includes VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstStageMask-03929
  If the geometry shaders feature is not enabled, dstStageMask must not contain
  VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstStageMask-03930
  If the tessellation shaders feature is not enabled, dstStageMask must not contain
  VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR or VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR

• VUID-VkBufferMemoryBarrier2KHR-dstStageMask-03931
  If the conditional rendering feature is not enabled, dstStageMask must not contain
  VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT
If the fragment density map feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT}.

If the transform feedback feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT}.

If the mesh shaders feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV}.

If the task shaders feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV}.

If the shading rate image feature is not enabled, \texttt{dstStageMask} must not contain \texttt{VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV}.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_INDEX_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_UNIFORM_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages.

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR}, \texttt{dstStageMask} must include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages.
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03907
  If dstAccessMask includes VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03908
  If dstAccessMask includes VK_ACCESS_2_SHADER_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03909
  If dstAccessMask includes VK_ACCESS_2_SHADER_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR, or one of the VK_PIPELINE_STAGE_*_SHADER_BIT stages

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03910
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03911
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03912
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03913
  If dstAccessMask includes VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03914
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03915
  If dstAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, dstStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR, VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR,
If `dstAccessMask` includes `VK_ACCESS_2_HOST_READ_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_HOST_WRITE_BIT_KHR`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_HOST_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV` or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

If `dstAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`, `dstStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`.
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03927
  If dstAccessMask includes VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-03928
  If dstAccessMask includes VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-04858
  If dstAccessMask includes VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-04859
  If dstAccessMask includes VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-04860
  If dstAccessMask includes VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-dstAccessMask-04861
  If dstAccessMask includes VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR, dstStageMask must include VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

- VUID-VkBufferMemoryBarrier2KHR-offset-01187
  offset must be less than the size of buffer

- VUID-VkBufferMemoryBarrier2KHR-size-01188
  If size is not equal to VK_WHOLE_SIZE, size must be greater than 0

- VUID-VkBufferMemoryBarrier2KHR-size-01189
  If size is not equal to VK_WHOLE_SIZE, size must be less than or equal to than the size of buffer minus offset

- VUID-VkBufferMemoryBarrier2KHR-buffer-04088
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-VkBufferMemoryBarrier2KHR-srcQueueFamilyIndex-04087
  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

- VUID-VkBufferMemoryBarrier2KHR-buffer-04089
  If buffer was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is a special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

- VUID-VkBufferMemoryBarrier2KHR-buffer-04090
  If buffer was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and
The `VkBufferMemoryBarrier` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkBufferMemoryBarrier;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
• `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
• `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
• `SrcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.
• `DstQueueFamilyIndex` is the destination queue family for a queue family ownership transfer.
• `buffer` is a handle to the buffer whose backing memory is affected by the barrier.
• `offset` is an offset in bytes into the backing memory for `buffer`; this is relative to the base offset as bound to the buffer (see `vkBindBufferMemory`).
• `size` is a size in bytes of the affected area of backing memory for `buffer`, or `VK_WHOLE_SIZE` to use the range from `offset` to the end of the buffer.

The first access scope is limited to access to memory through the specified buffer range, via access types in the source access mask specified by `srcAccessMask`. If `srcAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT`, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second access scope is limited to access to memory through the specified buffer range, via access types in the destination access mask specified by `dstAccessMask`. If `dstAccessMask` includes `VK_ACCESS_HOST_WRITE_BIT` or `VK_ACCESS_HOST_READ_BIT`, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, and `srcQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a queue family release operation for the specified buffer range, and the second access scope includes no access, as if `dstAccessMask` was 0.

If `dstQueueFamilyIndex` is not equal to `srcQueueFamilyIndex`, and `dstQueueFamilyIndex` is equal to the current queue family, then the memory barrier defines a queue family acquire operation for the specified buffer range, and the first access scope includes no access, as if `srcAccessMask` was 0.
Valid Usage

- **VUID-VkBufferMemoryBarrier-offset-01187**  
  *offset* must be less than the size of *buffer*

- **VUID-VkBufferMemoryBarrier-size-01188**  
  If *size* is not equal to VK_WHOLE_SIZE, *size* must be greater than 0

- **VUID-VkBufferMemoryBarrier-size-01189**  
  If *size* is not equal to VK_WHOLE_SIZE, *size* must be less than or equal to than the size of *buffer* minus *offset*

- **VUID-VkBufferMemoryBarrier-buffer-01931**  
  If *buffer* is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- **VUID-VkBufferMemoryBarrier-srcQueueFamilyIndex-04087**  
  If *srcQueueFamilyIndex* is not equal to *dstQueueFamilyIndex*, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

- **VUID-VkBufferMemoryBarrier-buffer-04088**  
  If *buffer* was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* are not equal, and one of *srcQueueFamilyIndex* and *dstQueueFamilyIndex* is a special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

- **VUID-VkBufferMemoryBarrier-buffer-04089**  
  If *buffer* was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and *srcQueueFamilyIndex* and *dstQueueFamilyIndex* are not equal, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer

- **VUID-VkBufferMemoryBarrier-synchronization2-03853**  
  If the synchronization2 feature is not enabled, and *buffer* was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, at least one of *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must be VK_QUEUE_FAMILY_IGNORED

Valid Usage (Implicit)

- **VUID-VkBufferMemoryBarrier-sType-sType**  
  *sType* must be VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER

- **VUID-VkBufferMemoryBarrier-pNext-pNext**  
  *pNext* must be NULL

- **VUID-VkBufferMemoryBarrier-buffer-parameter**  
  *buffer* must be a valid VkBuffer handle
7.7.3. Image Memory Barriers

Image memory barriers only apply to memory accesses involving a specific image subresource range. That is, a memory dependency formed from an image memory barrier is scoped to access via the specified image subresource range. Image memory barriers can also be used to define image layout transitions or a queue family ownership transfer for the specified image subresource range.

The `VkImageMemoryBarrier2KHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkImageMemoryBarrier2KHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkPipelineStageFlags2KHR    srcStageMask;
    VkAccessFlags2KHR           srcAccessMask;
    VkPipelineStageFlags2KHR    dstStageMask;
    VkAccessFlags2KHR           dstAccessMask;
    VkImageLayout               oldLayout;
    VkImageLayout               newLayout;
    uint32_t                    srcQueueFamilyIndex;
    uint32_t                    dstQueueFamilyIndex;
    VkImage                     image;
    VkImageSubresourceRange     subresourceRange;
} VkImageMemoryBarrier2KHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `srcStageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the first synchronization scope.
- `srcAccessMask` is a `VkAccessFlags2KHR` mask of pipeline first access scope.
- `dstStageMask` is a `VkPipelineStageFlags2KHR` mask of pipeline stages to be included in the second synchronization scope.
- `dstStageMask` is a `VkAccessFlags2KHR` mask of pipeline second access scope.
- `oldLayout` is the old layout in an image layout transition.
- `newLayout` is the new layout in an image layout transition.
- `srcQueueFamilyIndex` is the source queue family for a queue family ownership transfer.
- `dstQueueFamilyIndex` is the destination queue family for a queue family ownership transfer.
- `image` is a handle to the image affected by this barrier.
- `subresourceRange` describes the image subresource range within image that is affected by this barrier.

This structure defines a memory dependency limited to an image subresource range, and optionally defines a queue family transfer operation and image layout transition for that.
subresource range.

The first **synchronization scope** and **access scope** described by this structure include only operations and memory accesses specified by elements `srcStageMask` and `srcAccessMask`.

The second **synchronization scope** and **access scope** described by this structure include only operations and memory accesses specified by elements `dstStageMask` and `dstAccessMask`.

Both **access scopes** are limited to only memory accesses to `image` in the subresource range defined by `subresourceRange`.

If `image` was created with `VK_SHARING_MODE_EXCLUSIVE`, and `srcQueueFamilyIndex` is not equal to `dstQueueFamilyIndex`, this memory barrier defines a **queue family transfer operation**. When executed on a queue in the family identified by `srcQueueFamilyIndex`, this barrier defines a **queue family release operation** for the specified image subresource range, and the second synchronization and access scopes do not synchronize operations on that queue. When executed on a queue in the family identified by `dstQueueFamilyIndex`, this barrier defines a **queue family acquire operation** for the specified image subresource range, and the first synchronization and access scopes do not synchronize operations on that queue.

A **queue family transfer operation** is also defined if the values are not equal, and either is one of the special queue family values reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer. A **queue family release operation** is defined when `dstQueueFamilyIndex` is one of those values, and a **queue family acquire operation** is defined when `srcQueueFamilyIndex` is one of those values.

If `oldLayout` is not equal to `newLayout`, then the memory barrier defines an **image layout transition** for the specified image subresource range. If this memory barrier defines a **queue family transfer operation**, the layout transition is only executed once between the queues.

**Note**

When the old and new layout are equal, the layout values are ignored - data is preserved no matter what values are specified, or what layout the image is currently in.

If `image` has a multi-planar format and the image is **disjoint**, then including `VK_IMAGE_ASPECT_COLOR_BIT` in the `aspectMask` member of `subresourceRange` is equivalent to including `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, and (for three-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`. 
Valid Usage

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03929
  If the geometry shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03930
  If the tessellation shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03931
  If the conditional rendering feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03932
  If the fragment density map feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03933
  If the transform feedback feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03934
  If the mesh shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03935
  If the task shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`.

• VUID-VkImageMemoryBarrier2KHR-srcStageMask-03936
  If the shading rate image feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`.

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03900
  If `srcAccessMask` includes `VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03901
  If `srcAccessMask` includes `VK_ACCESS_2_INDEX_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03902
  If `srcAccessMask` includes `VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03903
  If `srcAccessMask` includes `VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`.
If `srcAccessMask` includes `VK_ACCESS_2_UNIFORM_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

If `srcAccessMask` includes `VK_ACCESS_2_SHADER_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`, or one of the `VK_PIPELINE_STAGE_*_SHADER_BIT` stages

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`

If `srcAccessMask` includes `VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`

If `srcAccessMask` includes `VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR`, `srcStageMask` must include `VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR`, `VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR`, or `VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR`
VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03914
  If srcAccessMask includes VK_ACCESS_2_TRANSFER_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03915
  If srcAccessMask includes VK_ACCESS_2_TRANSFER_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_COPY_BIT_KHR, VK_PIPELINE_STAGE_2_BLIT_BIT_KHR,
  VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR, VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR,
  VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR, VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03916
  If srcAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03917
  If srcAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03918
  If srcAccessMask includes VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03919
  If srcAccessMask includes VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03920
  If srcAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-04747
  If srcAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR, VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03922
  If srcAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT, VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

• VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03923
  If srcAccessMask includes VK_ACCESS_2_SHADING_RATE_IMAGE_READ_BIT_NV, srcStageMask must include
  VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV,
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03924
  If srcAccessMask includes VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV, srcStageMask must include
  VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03925
  If srcAccessMask includes VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV, srcStageMask must include
  VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03926
  If srcAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT, srcStageMask must include
  VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03927
  If srcAccessMask includes VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR, or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-03928
  If srcAccessMask includes VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR or
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-04858
  If srcAccessMask includes VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-04859
  If srcAccessMask includes VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-04860
  If srcAccessMask includes VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-srcAccessMask-04861
  If srcAccessMask includes VK_ACCESS_2_VIDEO_ENCODE_WRITE_BIT_KHR, srcStageMask must include
  VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstStageMask-03929
  If the geometry shaders feature is not enabled, dstStageMask must not contain
  VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstStageMask-03930
  If the tessellation shaders feature is not enabled, dstStageMask must not contain
  VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR or
  VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstStageMask-03931
  If the conditional rendering feature is not enabled, dstStageMask must not contain
  VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT
If the **fragment density map** feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT}

If the **transform feedback** feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT}

If the **mesh shaders** feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV}

If the **task shaders** feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV}

If the **shading rate image** feature is not enabled, \texttt{dstStageMask} \textbf{must} not contain \texttt{VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV}

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_INDIRECT_COMMAND_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_INDEX_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_INDEX_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_VERTEX_ATTRIBUTE_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_VERTEX_ATTRIBUTE_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_VERTEX_INPUT_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_INPUT_ATTACHMENT_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, or \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_UNIFORM_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_SHADER_SAMPLED_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR}, \texttt{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR}, or one of the \texttt{VK_PIPELINE_STAGE_*_SHADER_BIT} stages

If \texttt{dstAccessMask} includes \texttt{VK_ACCESS_2_SHADER_STORAGE_READ_BIT_KHR}, \texttt{dstStageMask} \textbf{must} include \texttt{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR},
If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_STORAGE_WRITE_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \), or one of the \( \text{VK_PIPELINE_STAGE_*_SHADER_BIT} \) stages.

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_READ_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR} \), or one of the \( \text{VK_PIPELINE_STAGE_*_SHADER_BIT} \) stages.

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_SHADER_WRITE_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \), or one of the \( \text{VK_PIPELINE_STAGE_*_SHADER_BIT} \) stages.

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_COLOR_ATTACHMENT_READ_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), or \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \).

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), or \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \).

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_READ_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), or \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \).

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_EARLY_FRAGMENT_TESTS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_LATE_FRAGMENT_TESTS_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR} \), or \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \).

If \( \text{dstAccessMask} \) includes \( \text{VK_ACCESS_2_TRANSFER_READ_BIT_KHR} \), \( \text{dstStageMask} \) must include \( \text{VK_PIPELINE_STAGE_2_COPY_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_BLIT_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_RESOLVE_BIT_KHR} \), \( \text{VK_PIPELINE_STAGE_2_CLEAR_BIT_KHR} \), or \( \text{VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR} \).
VK_PIPELINE_STAGE_2_ALL_TRANSFER_BIT_KHR,  
VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR,  
VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03916  
  If dstAccessMask includes VK_ACCESS_2_HOST_READ_BIT_KHR, dstStageMask must include  
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03917  
  If dstAccessMask includes VK_ACCESS_2_HOST_WRITE_BIT_KHR, dstStageMask must include  
  VK_PIPELINE_STAGE_2_HOST_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03918  
  If dstAccessMask includes VK_ACCESS_2CONDITIONAL_RENDERING_READ_BIT_EXT, dstStageMask  
  must include VK_PIPELINE_STAGE_2CONDITIONAL_RENDERING_BIT_EXT,  
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03919  
  If dstAccessMask includes VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT, dstStageMask  
  must include VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT,  
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03920  
  If dstAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT, dstStageMask  
  must include VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,  
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-04747  
  If dstAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT, dstStageMask  
  must include VK_PIPELINE_STAGE_2_DRAW_INDIRECT_BIT_KHR,  
  VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT,  
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03922  
  If dstAccessMask includes VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT, dstStageMask  
  must include VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV or  
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03923  
  If dstAccessMask includes VK_ACCESS_2_SHADING_RATE_IMAGE_READ_NONCOHERENT_BIT_EXT, dstStageMask must  
  include VK_PIPELINE_STAGE_2SHADING_RATE_IMAGE_BIT_NV,  
  VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03924  
  If dstAccessMask includes VK_ACCESS_2_COMMAND_PROCESS_READ_BIT_NV, dstStageMask must  
  include VK_PIPELINE_STAGE_2_COMMAND_PROCESS_READ_BIT_NV or  
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03925  
  If dstAccessMask includes VK_ACCESS_2_COMMAND_PROCESS_WRITE_BIT_NV, dstStageMask must  
  include VK_PIPELINE_STAGE_2_COMMAND_PROCESS_WRITE_BIT_NV or  
  VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR  

• VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03926  
  If dstAccessMask includes VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT, dstStageMask must  
  include VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT_KHR
VK_PIPELINE_STAGE_2_ALL_GRAPHICS_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03927
  If \( \text{dstAccessMask} \) includes VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR, \( \text{dstStageMask} \) must include VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR, VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR, or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-03928
  If \( \text{dstAccessMask} \) includes VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR, \( \text{dstStageMask} \) must include VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR or VK_PIPELINE_STAGE_2_ALL_COMMANDS_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-04858
  If \( \text{dstAccessMask} \) includes VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR, \( \text{dstStageMask} \) must include VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-dstAccessMask-04859
  If \( \text{dstAccessMask} \) includes VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR, \( \text{dstStageMask} \) must include VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR

- VUID-VkImageMemoryBarrier2KHR-subresourceRange-01486
  \( \text{subresourceRange.baseMipLevel} \) must be less than the \( \text{mipLevels} \) specified in \( \text{VkImageCreateInfo} \) when \( \text{image} \) was created

- VUID-VkImageMemoryBarrier2KHR-subresourceRange-01724
  If \( \text{subresourceRange.levelCount} \) is not VK_REMAINING_MIP_LEVELS, \( \text{subresourceRange.baseMipLevel} + \text{subresourceRange.levelCount} \) must be less than or equal to the \( \text{mipLevels} \) specified in \( \text{VkImageCreateInfo} \) when \( \text{image} \) was created

- VUID-VkImageMemoryBarrier2KHR-subresourceRange-01488
  \( \text{subresourceRange.baseArrayLayer} \) must be less than the \( \text{arrayLayers} \) specified in \( \text{VkImageCreateInfo} \) when \( \text{image} \) was created

- VUID-VkImageMemoryBarrier2KHR-subresourceRange-01725
  If \( \text{subresourceRange.layerCount} \) is not VK_REMAINING_ARRAY_LAYERS, \( \text{subresourceRange.baseArrayLayer} + \text{subresourceRange.layerCount} \) must be less than or equal to the \( \text{arrayLayers} \) specified in \( \text{VkImageCreateInfo} \) when \( \text{image} \) was created

- VUID-VkImageMemoryBarrier2KHR-image-01932
  If \( \text{image} \) is non-sparse then it must be bound completely and contiguously to a single \( \text{VkDeviceMemory} \) object

- VUID-VkImageMemoryBarrier2KHR-oldLayout-01208
  If \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) define a queue family ownership transfer or \( \text{oldLayout} \) and \( \text{newLayout} \) define an image layout transition, and \( \text{oldLayout} \) or \( \text{newLayout} \) is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL then \( \text{image} \) must have been created with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.
If \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) define a queue family ownership transfer or \( \text{oldLayout} \) and \( \text{newLayout} \) define an image layout transition, and \( \text{oldLayout} \) or \( \text{newLayout} \) is \( \text{VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR} \), image must have been created with \( \text{VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT} \) or \( \text{VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-srcQueueFamilyIndex-03939**
  If \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) define a queue family ownership transfer or \( \text{oldLayout} \) and \( \text{newLayout} \) define an image layout transition, and \( \text{oldLayout} \) or \( \text{newLayout} \) is \( \text{VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR} \), image must have been created with at least one of \( \text{VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT} \), \( \text{VK_IMAGE_USAGE_SAMPLED_BIT} \), or \( \text{VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-oldLayout-02088**
  If \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) define a queue family ownership transfer or \( \text{oldLayout} \) and \( \text{newLayout} \) define an image layout transition, and \( \text{oldLayout} \) or \( \text{newLayout} \) is \( \text{VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR} \) then image must have been created with \( \text{VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR} \).

- **VUID-VkImageMemoryBarrier2KHR-image-01671**
  If image has a single-plane color format or is not disjoint, then the \( \text{aspectMask} \) member of \( \text{subresourceRange} \) must be \( \text{VK_IMAGE_ASPECT_COLOR_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-image-01672**
  If image has a multi-planar format and the image is disjoint, then the \( \text{aspectMask} \) member of \( \text{subresourceRange} \) must include either at least one of \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \), \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \), and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \); or \( \text{aspectMask} \) must include \( \text{VK_IMAGE_ASPECT_COLOR_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-image-01673**
  If image has a multi-planar format with only two planes, then the \( \text{aspectMask} \) member of \( \text{subresourceRange} \) must not include \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-image-03319**
  If image has a depth/stencil format with both depth and stencil and the \( \text{separateDepthStencilLayouts} \) feature is enabled, then the \( \text{aspectMask} \) member of \( \text{subresourceRange} \) must include either or both \( \text{VK_IMAGE_ASPECT_DEPTH_BIT} \) and \( \text{VK_IMAGE_ASPECT_STENCIL_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-image-03320**
  If image has a depth/stencil format with both depth and stencil and the \( \text{separateDepthStencilLayouts} \) feature is not enabled, then the \( \text{aspectMask} \) member of \( \text{subresourceRange} \) must include both \( \text{VK_IMAGE_ASPECT_DEPTH_BIT} \) and \( \text{VK_IMAGE_ASPECT_STENCIL_BIT} \).

- **VUID-VkImageMemoryBarrier2KHR-srcQueueFamilyIndex-04070**
  If \( \text{srcQueueFamilyIndex} \) is not equal to \( \text{dstQueueFamilyIndex} \), at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer.

- **VUID-VkImageMemoryBarrier2KHR-image-04071**
  If image was created with a sharing mode of \( \text{VK_SHARING_MODE_CONCURRENT} \), \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) are not equal, and one of \( \text{srcQueueFamilyIndex} \) and \( \text{dstQueueFamilyIndex} \) is a special queue family values reserved for external memory transfers, the other must be \( \text{VK_QUEUE_FAMILY_IGNORED} \).
If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex and dstQueueFamilyIndex must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer.

If srcStageMask or dstStageMask include VK_PIPELINE_STAGE_2_HOST_BIT_KHR, srcQueueFamilyIndex and dstQueueFamilyIndex must be equal.

If srcStageMask includes VK_PIPELINE_STAGE_2_HOST_BIT_KHR, and srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define a image layout transition, oldLayout must be one of VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_UNDEFINED, or VK_IMAGE_LAYOUT_GENERAL.

Valid Usage (Implicit)

sType must be VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER_2_KHR

pNext must be NULL or a pointer to a valid instance of VkSampleLocationsInfoEXT

The sType value of each struct in the pNext chain must be unique

srcStageMask must be a valid combination of VkPipelineStageFlagBits2KHR values

dstStageMask must be a valid combination of VkPipelineStageFlagBits2KHR values

srcAccessMask must be a valid combination of VkAccessFlagBits2KHR values

dstAccessMask must be a valid combination of VkAccessFlagBits2KHR values

oldLayout must be a valid VkImageLayout value

newLayout must be a valid VkImageLayout value

image must be a valid VkImage handle

subresourceRange must be a valid VkImageSubresourceRange structure

The VkImageMemoryBarrier structure is defined as:
typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcAccessMask** is a bitmask of VkAccessFlagBits specifying a source access mask.
- **dstAccessMask** is a bitmask of VkAccessFlagBits specifying a destination access mask.
- **oldLayout** is the old layout in an image layout transition.
- **newLayout** is the new layout in an image layout transition.
- **srcQueueFamilyIndex** is the source queue family for a queue family ownership transfer.
- **dstQueueFamilyIndex** is the destination queue family for a queue family ownership transfer.
- **image** is a handle to the image affected by this barrier.
- **subresourceRange** describes the image subresource range within image that is affected by this barrier.

The first access scope is limited to access to memory through the specified image subresource range, via access types in the source access mask specified by srcAccessMask. If srcAccessMask includes VK_ACCESS_HOST_WRITE_BIT, memory writes performed by that access type are also made visible, as that access type is not performed through a resource.

The second access scope is limited to access to memory through the specified image subresource range, via access types in the destination access mask specified by dstAccessMask. If dstAccessMask includes VK_ACCESS_HOST_WRITE_BIT or VK_ACCESS_HOST_READ_BIT, available memory writes are also made visible to accesses of those types, as those access types are not performed through a resource.

If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, and srcQueueFamilyIndex is equal to the current queue family, then the memory barrier defines a queue family release operation for the specified image subresource range, and the second access scope includes no access, as if dstAccessMask was 0.

If dstQueueFamilyIndex is not equal to srcQueueFamilyIndex, and dstQueueFamilyIndex is equal to the current queue family, then the memory barrier defines a queue family acquire operation for the specified image subresource range, and the first access scope includes no access, as if srcAccessMask was 0.
If the **synchronization2 feature** is not enabled or **oldLayout** is not equal to **newLayout**, **oldLayout** and **newLayout** define an **image layout transition** for the specified image subresource range.

**Note**

If the **synchronization2 feature** is enabled, when the old and new layout are equal, the layout values are ignored - data is preserved no matter what values are specified, or what layout the image is currently in.

If **image** has a multi-planar format and the image is **disjoint**, then including **VK_IMAGE_ASPECT_COLOR_BIT** in the **aspectMask** member of **subresourceRange** is equivalent to including **VK_IMAGE_ASPECT_PLANE_0_BIT**, **VK_IMAGE_ASPECT_PLANE_1_BIT**, and (for three-plane formats only) **VK_IMAGE_ASPECT_PLANE_2_BIT**.
Valid Usage

• VUID-VkImageMemoryBarrier-subresourceRange-01486
  subresourceRange.baseMipLevel must be less than the mipLevels specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-subresourceRange-01724
  If subresourceRange.levelCount is not VK_REMAINING_MIP_LEVELS, subresourceRange.baseMipLevel + subresourceRange.levelCount must be less than or equal to the mipLevels specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-subresourceRange-01488
  subresourceRange.baseArrayLayer must be less than the arrayLayers specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-subresourceRange-01725
  If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the arrayLayers specified in VkImageCreateInfo when image was created

• VUID-VkImageMemoryBarrier-image-01932
  If image is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-VkImageMemoryBarrier-oldLayout-01208
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL then image must have been created with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier-oldLayout-01209
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL then image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier-oldLayout-01210
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL then image must have been created with VK_IMAGE_USAGE_SAMPLED_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• VUID-VkImageMemoryBarrier-oldLayout-01211
  If srcQueueFamilyIndex and dstQueueFamilyIndex define a queue family ownership transfer or oldLayout and newLayout define an image layout transition, and oldLayout or newLayout is VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL then image must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT
If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT`.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, `oldLayout` must be `VK_IMAGE_LAYOUT_UNDEFINED` or the current layout of the image subresources affected by the barrier.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, `newLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` then image must have been created with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR`, image must have been created with `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` or `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR`, image must have been created with at least one of `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_SAMPLED_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

If `srcQueueFamilyIndex` and `dstQueueFamilyIndex` define a queue family ownership transfer or `oldLayout` and `newLayout` define an image layout transition, and `oldLayout` or `newLayout` is `VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR` then image must have been created with `VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR` set.

If image has a single-plane color format or is not disjoint, then the aspectMask member of subresourceRange must be `VK_IMAGE_ASPECT_COLOR_BIT`.

If image has a multi-planar format and the image is disjoint, then the aspectMask member

---

372 | Chapter 7. Synchronization and Cache Control
of subresourceRange must include either at least one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, and VK_IMAGE_ASPECT_PLANE_2_BIT; or must include VK_IMAGE_ASPECT_COLOR_BIT

• VUID-VkImageMemoryBarrier-image-01673

  If image has a multi-planar format with only two planes, then the aspectMask member of subresourceRange must not include VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-VkImageMemoryBarrier-image-03319

  If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is enabled, then the aspectMask member of subresourceRange must include either or both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier-image-03320

  If image has a depth/stencil format with both depth and stencil and the separateDepthStencilLayouts feature is not enabled, then the aspectMask member of subresourceRange must include both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04070

  If srcQueueFamilyIndex is not equal to dstQueueFamilyIndex, at least one must not be a special queue family reserved for external memory ownership transfers, as described in Queue Family Ownership Transfer

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04071

  If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, and one of srcQueueFamilyIndex and dstQueueFamilyIndex is a special queue family values reserved for external memory transfers, the other must be VK_QUEUE_FAMILY_IGNORED

• VUID-VkImageMemoryBarrier-srcQueueFamilyIndex-04072

  If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not equal, srcQueueFamilyIndex and dstQueueFamilyIndex must both be valid queue families, or one of the special queue family values reserved for external memory transfers, as described in Queue Family Ownership Transfer

• VUID-VkImageMemoryBarrier-synchronization2-03857

  If the synchronization2 feature is not enabled, and image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, at least one of srcQueueFamilyIndex and dstQueueFamilyIndex must be VK_QUEUE_FAMILY_IGNORED
Valid Usage (Implicit)

- VUID-VkImageMemoryBarrier-sType-sType
  *sType* must be *VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER*

- VUID-VkImageMemoryBarrier-pNext-pNext
  *pNext* must be *NULL* or a pointer to a valid instance of *VkSampleLocationsInfoEXT*

- VUID-VkImageMemoryBarrier-sType-unique
  The *sType* value of each struct in the *pNext* chain must be unique

- VUID-VkImageMemoryBarrier-oldLayout-parameter
  *oldLayout* must be a valid *VkImageLayout* value

- VUID-VkImageMemoryBarrier-newLayout-parameter
  *newLayout* must be a valid *VkImageLayout* value

- VUID-VkImageMemoryBarrier-image-parameter
  *image* must be a valid *VkImage* handle

- VUID-VkImageMemoryBarrier-subresourceRange-parameter
  *subresourceRange* must be a valid *VkImageSubresourceRange* structure

7.7.4. Queue Family Ownership Transfer

Resources created with a *VkSharingMode* of *VK_SHARING_MODE_EXCLUSIVE* must have their ownership explicitly transferred from one queue family to another in order to access their content in a well-defined manner on a queue in a different queue family. Resources shared with external APIs or instances using external memory must also explicitly manage ownership transfers between local and external queues (or equivalent constructs in external APIs) regardless of the *VkSharingMode* specified when creating them. The special queue family index *VK_QUEUE_FAMILY_EXTERNAL* represents any queue external to the resource’s current Vulkan instance, as long as the queue uses the same underlying physical device or device group and uses the same driver version as the resource’s *VkDevice*, as indicated by *VkPhysicalDeviceIDProperties::deviceUUID* and *VkPhysicalDeviceIDProperties::driverUUID*. The special queue family index *VK_QUEUE_FAMILY_FOREIGN_EXT* represents any queue external to the resource’s current Vulkan instance, regardless of the queue’s underlying physical device or driver version. This includes, for example, queues for fixed-function image processing devices, media codec devices, and display devices, as well as all queues that use the same underlying physical device (or device group) and driver version as the resource’s *VkDevice*. If memory dependencies are correctly expressed between uses of such a resource between two queues in different families, but no ownership transfer is defined, the contents of that resource are undefined for any read accesses performed by the second queue family.

*Note*

If an application does not need the contents of a resource to remain valid when transferring from one queue family to another, then the ownership transfer *should* be skipped.
Applications should expect transfers to/from `VK_QUEUE_FAMILY_FOREIGN_EXT` to be more expensive than transfers to/from `VK_QUEUE_FAMILY_EXTERNAL_KHR`.

A queue family ownership transfer consists of two distinct parts:

1. Release exclusive ownership from the source queue family
2. Acquire exclusive ownership for the destination queue family

An application **must** ensure that these operations occur in the correct order by defining an execution dependency between them, e.g. using a semaphore.

A *release operation* is used to release exclusive ownership of a range of a buffer or image subresource range. A release operation is defined by executing a *buffer memory barrier* (for a buffer range) or an *image memory barrier* (for an image subresource range) using `vkCmdPipelineBarrier`, on a queue from the source queue family. The `srcQueueFamilyIndex` parameter of the barrier **must** be set to the source queue family index, and the `dstQueueFamilyIndex` parameter to the destination queue family index. `dstAccessMask` is ignored for such a barrier, such that no visibility operation is executed - the value of this mask does not affect the validity of the barrier. The release operation happens-after the availability operation, and happens-before operations specified in the second synchronization scope of the calling command.

An *acquire operation* is used to acquire exclusive ownership of a range of a buffer or image subresource range. An acquire operation is defined by executing a *buffer memory barrier* (for a buffer range) or an *image memory barrier* (for an image subresource range) using `vkCmdPipelineBarrier`, on a queue from the destination queue family. The buffer range or image subresource range specified in an acquire operation **must** match exactly that of a previous release operation. The `srcQueueFamilyIndex` parameter of the barrier **must** be set to the source queue family index, and the `dstQueueFamilyIndex` parameter to the destination queue family index. `srcAccessMask` is ignored for such a barrier, such that no availability operation is executed - the value of this mask does not affect the validity of the barrier. The acquire operation happens-after operations in the first synchronization scope of the calling command, and happens-before the visibility operation.

If the transfer is via an image memory barrier, and an *image layout transition* is desired, then the values of `oldLayout` and `newLayout` in the *release operation*’s memory barrier **must** be equal to values of `oldLayout` and `newLayout` in the *acquire operation*’s memory barrier. Although the image layout transition is submitted twice, it will only be executed once. A layout transition specified in
this way happens-after the release operation and happens-before the acquire operation.

If the values of srcQueueFamilyIndex and dstQueueFamilyIndex are equal, no ownership transfer is performed, and the barrier operates as if they were both set to VK_QUEUE_FAMILY_IGNORED.

Queue family ownership transfers may perform read and write accesses on all memory bound to the image subresource or buffer range, so applications must ensure that all memory writes have been made available before a queue family ownership transfer is executed. Available memory is automatically made visible to queue family release and acquire operations, and writes performed by those operations are automatically made available.

Once a queue family has acquired ownership of a buffer range or image subresource range of a VK_SHARING_MODE_EXCLUSIVE resource, its contents are undefined to other queue families unless ownership is transferred. The contents of any portion of another resource which aliases memory that is bound to the transferred buffer or image subresource range are undefined after a release or acquire operation.

Note
Because events cannot be used directly for inter-queue synchronization, and because vkCmdSetEvent does not have the queue family index or memory barrier parameters needed by a release operation, the release and acquire operations of a queue family ownership transfer can only be performed using vkCmdPipelineBarrier.

7.8. Wait Idle Operations

To wait on the host for the completion of outstanding queue operations for a given queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueWaitIdle(
    VkQueue queue);
```

- `queue` is the queue on which to wait.

`vkQueueWaitIdle` is equivalent to submitting a fence to a queue and waiting with an infinite timeout for that fence to signal.

Valid Usage (Implicit)

- `VUID-vkQueueWaitIdle-queue-parameter queue must be a valid VkQueue handle`

Host Synchronization

- Host access to `queue` must be externally synchronized
Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>

Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

To wait on the host for the completion of outstanding queue operations for all queues on a given logical device, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkDeviceWaitIdle(
    VkDevice device);
```

- `device` is the logical device to idle.

`vkDeviceWaitIdle` is equivalent to calling `vkQueueWaitIdle` for all queues owned by `device`.

Valid Usage (Implicit)

- VUID-vkDeviceWaitIdle-device-parameter
  
  `device` must be a valid `VkDevice` handle

Host Synchronization

- Host access to all `VkQueue` objects created from `device` must be externally synchronized
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

7.9. Host Write Ordering Guarantees

When batches of command buffers are submitted to a queue via a queue submission command, it defines a memory dependency with prior host operations, and execution of command buffers submitted to the queue.

The first synchronization scope is defined by the host execution model, but includes execution of vkQueueSubmit on the host and anything that happened-before it.

The second synchronization scope includes all commands submitted in the same queue submission, and all commands that occur later in submission order.

The first access scope includes all host writes to mappable device memory that are available to the host memory domain.

The second access scope includes all memory access performed by the device.

7.10. Synchronization and Multiple Physical Devices

If a logical device includes more than one physical device, then fences, semaphores, and events all still have a single instance of the signaled state.

A fence becomes signaled when all physical devices complete the necessary queue operations.

Semaphore wait and signal operations all include a device index that is the sole physical device that performs the operation. These indices are provided in the VkDeviceGroupSubmitInfo and VkDeviceGroupBindSparseInfo structures. Semaphores are not exclusively owned by any physical device. For example, a semaphore can be signaled by one physical device and then waited on by a different physical device.

An event can only be waited on by the same physical device that signaled it (or the host).

7.11. Calibrated timestamps

In order to be able to correlate the time a particular operation took place at on timelines of different time domains (e.g. a device operation vs a host operation), Vulkan allows querying
calibrated timestamps from multiple time domains.

To query calibrated timestamps from a set of time domains, call:

```c
// Provided by VK_EXT_calibrated_timestamps
VkResult vkGetCalibratedTimestampsEXT(
    VkDevice device,                        // Logical device
    uint32_t timestampCount,               // Number of timestamps to query
    const VkCalibratedTimestampInfoEXT* pTimestampInfos,  // Pointer to array
    uint64_t* pTimestamps,                 // Pointer to output array
    uint64_t* pMaxDeviation);             // Pointer to max deviation
```

- **device** is the logical device used to perform the query.
- **timestampCount** is the number of timestamps to query.
- **pTimestampInfos** is a pointer to an array of **timestampCount** `VkCalibratedTimestampInfoEXT` structures, describing the time domains the calibrated timestamps should be captured from.
- **pTimestamps** is a pointer to an array of **timestampCount** 64-bit unsigned integer values in which the requested calibrated timestamp values are returned.
- **pMaxDeviation** is a pointer to a 64-bit unsigned integer value in which the strictly positive maximum deviation, in nanoseconds, of the calibrated timestamp values is returned.

**Note**

The maximum deviation **may** vary between calls to `vkGetCalibratedTimestampsEXT` even for the same set of time domains due to implementation and platform specific reasons. It is the application’s responsibility to assess whether the returned maximum deviation makes the timestamp values suitable for any particular purpose and **can** choose to re-issue the timestamp calibration call pursuing a lower deviation value.

Calibrated timestamp values **can** be extrapolated to estimate future coinciding timestamp values, however, depending on the nature of the time domains and other properties of the platform extrapolating values over a sufficiently long period of time **may** no longer be accurate enough to fit any particular purpose, so applications are expected to re-calibrate the timestamps on a regular basis.
Valid Usage (Implicit)

- VUID-vkGetCalibratedTimestampsEXT-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetCalibratedTimestampsEXT-pTimestampInfos-parameter
pTimestampInfos must be a valid pointer to an array of timestampCount valid VkCalibratedTimestampInfoEXT structures

- VUID-vkGetCalibratedTimestampsEXT-pTimestamps-parameter
pTimestamps must be a valid pointer to an array of timestampCount uint64_t values

- VUID-vkGetCalibratedTimestampsEXT-pMaxDeviation-parameter
pMaxDeviation must be a valid pointer to a uint64_t value

- VUID-vkGetCalibratedTimestampsEXT-timestampCount-arraylength
timestampCount must be greater than 0

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkCalibratedTimestampInfoEXT structure is defined as:

```
// Provided by VK_EXT_calibrated_timestamps
typedef struct VkCalibratedTimestampInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkTimeDomainEXT    timeDomain;
} VkCalibratedTimestampInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **timeDomain** is a VkTimeDomainEXT value specifying the time domain from which the calibrated timestamp value should be returned.
Valid Usage

- VUID-VkCalibratedTimestampInfoEXT-timeDomain-02354
timeDomain must be one of the VkTimeDomainEXT values returned by vkGetPhysicalDeviceCalibrateableTimeDomainsEXT

Valid Usage (Implicit)

- VUID-VkCalibratedTimestampInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT

- VUID-VkCalibratedTimestampInfoEXT-pNext-pNext
  pNext must be NULL

- VUID-VkCalibratedTimestampInfoEXT-timeDomain-parameter
timeDomain must be a valid VkTimeDomainEXT value

The set of supported time domains consists of:

```
// Provided by VK_EXT_calibrated_timestamps
typedef enum VkTimeDomainEXT {
  VK_TIME_DOMAIN_DEVICE_EXT = 0,
  VK_TIME_DOMAIN_CLOCK_MONOTONIC_EXT = 1,
  VK_TIME_DOMAIN_CLOCK_MONOTONIC_RAW_EXT = 2,
  VK_TIME_DOMAIN_QUERY_PERFORMANCE_COUNTER_EXT = 3,
} VkTimeDomainEXT;
```

- **VK_TIME_DOMAIN_DEVICE_EXT** specifies the device time domain. Timestamp values in this time domain use the same units and are comparable with device timestamp values captured using vkCmdWriteTimestamp or vkCmdWriteTimestamp2KHR and are defined to be incrementing according to the timestampPeriod of the device.

- **VK_TIME_DOMAIN_CLOCK_MONOTONIC_EXT** specifies the CLOCK_MONOTONIC time domain available on POSIX platforms. Timestamp values in this time domain are in units of nanoseconds and are comparable with platform timestamp values captured using the POSIX clock_gettime API as computed by this example:

```
Note

An implementation supporting VK_EXT_calibrated_timestamps will use the same time domain for all its VkQueue so that timestamp values reported for VK_TIME_DOMAIN_DEVICE_EXT can be matched to any timestamp captured through vkCmdWriteTimestamp or vkCmdWriteTimestamp2KHR.
```
• **VK_TIME_DOMAIN_CLOCK_MONOTONIC_RAW_EXT** specifies the CLOCK_MONOTONIC_RAW time domain available on POSIX platforms. Timestamp values in this time domain are in units of nanoseconds and are comparable with platform timestamp values captured using the POSIX clock_gettime API as computed by this example:

```c
struct timespec tv;
clock_gettime(CLOCK_MONOTONIC, &tv);
return tv.tv_nsec + tv.tv_sec*1000000000ull;
```

• **VK_TIME_DOMAIN_QUERY_PERFORMANCE_COUNTER_EXT** specifies the performance counter (QPC) time domain available on Windows. Timestamp values in this time domain are in the same units as those provided by the Windows QueryPerformanceCounter API and are comparable with platform timestamp values captured using that API as computed by this example:

```c
LARGE_INTEGER counter;
QueryPerformanceCounter(&counter);
return counter.QuadPart;
```
Chapter 8. Render Pass

A render pass represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses. The use of a render pass in a command buffer is a render pass instance.

Render passes are represented by VkRenderPass handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkRenderPass)
```

An attachment description describes the properties of an attachment including its format, sample count, and how its contents are treated at the beginning and end of each render pass instance.

A subpass represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular subpass of a render pass instance.

A subpass description describes the subset of attachments that is involved in the execution of a subpass. Each subpass can read from some attachments as input attachments, write to some as color attachments or depth/stencil attachments, perform shader resolve operations to color_attachments or depth/stencil_attachments, and perform multisample resolve operations to resolve attachments. A subpass description can also include a set of preserve attachments, which are attachments that are not read or written by the subpass but whose contents must be preserved throughout the subpass.

A subpass uses an attachment if the attachment is a color, depth/stencil, resolve, depth/stencil resolve, fragment shading rate, or input attachment for that subpass (as determined by the pColorAttachments, pDepthStencilAttachment, pResolveAttachments, VkSubpassDescriptionDepthStencilResolve::pDepthStencilResolveAttachment, VkFragmentShadingRateAttachmentInfoKHR, and pInputAttachments members of VkSubpassDescription, respectively). A subpass does not use an attachment if that attachment is preserved by the subpass. The first use of an attachment is in the lowest numbered subpass that uses that attachment. Similarly, the last use of an attachment is in the highest numbered subpass that uses that attachment.

The subpasses in a render pass all render to the same dimensions, and fragments for pixel (x,y,layer) in one subpass can only read attachment contents written by previous subpasses at that same (x,y,layer) location. For multi-pixel fragments, the pixel read from an input attachment is selected from the pixels covered by that fragment in an implementation-dependent manner. However, this selection must be made consistently for any fragment with the same shading rate for the lifetime of the VkDevice.
Note

By describing a complete set of subpasses in advance, render passes provide the implementation an opportunity to optimize the storage and transfer of attachment data between subpasses.

In practice, this means that subpasses with a simple framebuffer-space dependency may be merged into a single tiled rendering pass, keeping the attachment data on-chip for the duration of a render pass instance. However, it is also quite common for a render pass to only contain a single subpass.

Subpass dependencies describe execution and memory dependencies between subpasses.

A subpass dependency chain is a sequence of subpass dependencies in a render pass, where the source subpass of each subpass dependency (after the first) equals the destination subpass of the previous dependency.

Execution of subpasses may overlap or execute out of order with regards to other subpasses, unless otherwise enforced by an execution dependency. Each subpass only respects submission order for commands recorded in the same subpass, and the vkCmdBeginRenderPass and vkCmdEndRenderPass commands that delimit the render pass - commands within other subpasses are not included. This affects most other implicit ordering guarantees.

A render pass describes the structure of subpasses and attachments independent of any specific image views for the attachments. The specific image views that will be used for the attachments, and their dimensions, are specified in VkFramebuffer objects. Framebuffers are created with respect to a specific render pass that the framebuffer is compatible with (see Render Pass Compatibility). Collectively, a render pass and a framebuffer define the complete render target state for one or more subpasses as well as the algorithmic dependencies between the subpasses.

The various pipeline stages of the drawing commands for a given subpass may execute concurrently and/or out of order, both within and across drawing commands, whilst still respecting pipeline order. However for a given (x,y,layer,sample) sample location, certain per-sample operations are performed in rasterization order.

8.1. Render Pass Creation

To create a render pass, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateRenderPass(
    VkDevice device,  // Provided by VK_VERSION_1_0
    const VkRenderPassCreateInfo* pCreateInfo,  // Provided by VK_VERSION_1_0
    const VkAllocationCallbacks* pAllocator,  // Provided by VK_VERSION_1_0
    VkRenderPass* pRenderPass);  // Provided by VK_VERSION_1_0
```

- `device` is the logical device that creates the render pass.
- `pCreateInfo` is a pointer to a VkRenderPassCreateInfo structure describing the parameters of the
render pass.

• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

• **pRenderPass** is a pointer to a *VkRenderPass* handle in which the resulting render pass object is returned.

---

### Valid Usage (Implicit)

- **VUID-vkCreateRenderPass-device-parameter**
  
  *device* must be a valid *VkDevice* handle

- **VUID-vkCreateRenderPass-pCreateInfo-parameter**
  
  *pCreateInfo* must be a valid pointer to a valid *VkRenderPassCreateInfo* structure

- **VUID-vkCreateRenderPass-pAllocator-parameter**
  
  If *pAllocator* is not *NULL*, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkCreateRenderPass-pRenderPass-parameter**
  
  *pRenderPass* must be a valid pointer to a *VkRenderPass* handle

---

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The *VkRenderPassCreateInfo* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkRenderPassCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkRenderPassCreateFlags flags;
  uint32_t attachmentCount;
  const VkAttachmentDescription* pAttachments;
  uint32_t subpassCount;
  const VkSubpassDescription* pSubpasses;
  uint32_t dependencyCount;
  const VkSubpassDependency* pDependencies;
} VkRenderPassCreateInfo;
```

- **sType** is the type of this structure.

- **pNext** is *NULL* or a pointer to a structure extending this structure.
• **flags** is a bitmask of `VkRenderPassCreateFlagBits`.

• **attachmentCount** is the number of attachments used by this render pass.

• **pAttachments** is a pointer to an array of `attachmentCount` `VkAttachmentDescription` structures describing the attachments used by the render pass.

• **subpassCount** is the number of subpasses to create.

• **pSubpasses** is a pointer to an array of `subpassCount` `VkSubpassDescription` structures describing each subpass.

• **dependencyCount** is the number of memory dependencies between pairs of subpasses.

• **pDependencies** is a pointer to an array of `dependencyCount` `VkSubpassDependency` structures describing dependencies between pairs of subpasses.

**Note**

Care should be taken to avoid a data race here; if any subpasses access attachments with overlapping memory locations, and one of those accesses is a write, a subpass dependency needs to be included between them.
Valid Usage

- **VUID-VkRenderPassCreateInfo-attachment-00834**
  If the attachment member of any element of pInputAttachments, pColorAttachments, pResolveAttachments or pDepthStencilAttachment, or any element of pPreserveAttachments in any element of pSubpasses is not VK_ATTACHMENT_UNUSED, it must be less than attachmentCount.

- **VUID-VkRenderPassCreateInfo-pAttachments-00836**
  For any member of pAttachments with a loadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pAttachments-02511**
  For any member of pAttachments with a stencilLoadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pAttachments-01566**
  For any member of pAttachments with a loadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pAttachments-01567**
  For any member of pAttachments with a stencilLoadOp equal to VK_ATTACHMENT_LOAD_OP_CLEAR, the first use of that attachment must not specify a layout equal to VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkRenderPassCreateInfo-pNext-01926**
  If the pNext chain includes a VkRenderPassInputAttachmentAspectCreateInfo structure, the subpass member of each element of its pAspectReferences member must be less than subpassCount.

- **VUID-VkRenderPassCreateInfo-pNext-01963**
  If the pNext chain includes a VkRenderPassInputAttachmentAspectCreateInfo structure, for any element of the pInputAttachments member of any element of pSubpasses where the attachment member is not VK_ATTACHMENT_UNUSED, the aspectMask member of the corresponding element of VkRenderPassInputAttachmentAspectCreateInfo::pAspectReferences must only include aspects that are present in images of the format specified by the element of pAttachments at attachment.

- **VUID-VkRenderPassCreateInfo-pNext-01928**
  If the pNext chain includes a VkRenderPassMultiviewCreateInfo structure, and its subpassCount member is not zero, that member must be equal to the value of subpassCount.
If the `pNext` chain includes a VkRenderPassMultiviewCreateInfo structure, if its dependencyCount member is not zero, it must be equal to dependencyCount

- VUID-VkRenderPassCreateInfo-pNext-01930
  If the `pNext` chain includes a VkRenderPassMultiviewCreateInfo structure, for each non-zero element of pViewOffsets, the srcSubpass and dstSubpass members of pDependencies at the same index must not be equal

- VUID-VkRenderPassCreateInfo-pNext-02512
  If the `pNext` chain includes a VkRenderPassMultiviewCreateInfo structure, for each non-zero element of pViewOffsets, the srcSubpass and dstSubpass members of pDependencies at the same index must be 0

- VUID-VkRenderPassCreateInfo-pNext-02513
  If the `pNext` chain includes a VkRenderPassMultiviewCreateInfo structure, for any element of pDependencies with a dependencyFlags member that does not include VK_DEPENDENCY_VIEW_LOCAL_BIT, the corresponding element of the pViewOffsets member of that VkRenderPassMultiviewCreateInfo instance must be 0

- VUID-VkRenderPassCreateInfo-pNext-02514
  If the `pNext` chain includes a VkRenderPassMultiviewCreateInfo structure, for any element of pDependencies with a dependencyFlags member that does not include VK_DEPENDENCY_VIEW_LOCAL_BIT, the corresponding element of the pViewOffsets member of that VkRenderPassMultiviewCreateInfo instance must be 0

- VUID-VkRenderPassCreateInfo-pNext-02515
  If the `pNext` chain includes a VkRenderPassMultiviewCreateInfo structure, for any element of pDependencies with a dependencyFlags member that does not include VK_DEPENDENCY_VIEW_LOCAL_BIT, the corresponding element of the pViewOffsets member of that VkRenderPassMultiviewCreateInfo instance must be 0

- VUID-VkRenderPassCreateInfo-pDependencies-00837
  For any element of pDependencies, if the srcSubpass is not VK_SUBPASS_EXTERNAL, all stage flags included in the srcStageMask member of that dependency must be a pipeline stage supported by the pipeline identified by the pipelineBindPoint member of the source subpass

- VUID-VkRenderPassCreateInfo-pDependencies-00838
  For any element of pDependencies, if the dstSubpass is not VK_SUBPASS_EXTERNAL, all stage flags included in the dstStageMask member of that dependency must be a pipeline stage supported by the pipeline identified by the pipelineBindPoint member of the destination subpass

- VUID-VkRenderPassCreateInfo-srcSubpass-02517
  The srcSubpass member of each element of pDependencies must be less than subpassCount

- VUID-VkRenderPassCreateInfo-dstSubpass-02518
  The dstSubpass member of each element of pDependencies must be less than subpassCount
Valid Usage (Implicit)

- **VUID-VkRenderPassCreateInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO`

- **VUID-VkRenderPassCreateInfo-pNext-pNext**
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkRenderPassFragmentDensityMapCreateInfoEXT`, `VkRenderPassInputAttachmentAspectCreateInfo`, or `VkRenderPassMultiviewCreateInfo`

- **VUID-VkRenderPassCreateInfo-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkRenderPassCreateInfo-flags-parameter**
  - `flags` must be a valid combination of `VkRenderPassCreateFlagBits` values

- **VUID-VkRenderPassCreateInfo-pAttachments-parameter**
  - If `attachmentCount` is not 0, `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkAttachmentDescription` structures

- **VUID-VkRenderPassCreateInfo-pSubpasses-parameter**
  - `pSubpasses` must be a valid pointer to an array of `subpassCount` valid `VkSubpassDescription` structures

- **VUID-VkRenderPassCreateInfo-pDependencies-parameter**
  - If `dependencyCount` is not 0, `pDependencies` must be a valid pointer to an array of `dependencyCount` valid `VkSubpassDependency` structures

- **VUID-VkRenderPassCreateInfo-subpassCount-arraylength**
  - `subpassCount` must be greater than 0

Bits which **can** be set in `VkRenderPassCreateInfo::flags` describing additional properties of the render pass are:

```
// Provided by VK_VERSION_1_0
typedef enum VkRenderPassCreateFlagBits {
    // Provided by VK_QCOM_render_pass_transform
    VK_RENDER_PASS_CREATE_TRANSFORM_BIT_QCOM = 0x00000002,
} VkRenderPassCreateFlagBits;
```

- **VK_RENDER_PASS_CREATE_TRANSFORM_BIT_QCOM** specifies that the created renderpass is compatible with render pass transform.

```
// Provided by VK_VERSION_1_0
typedef VkFlags VkRenderPassCreateFlags;
```

`VkRenderPassCreateFlags` is a bitmask type for setting a mask of zero or more `VkRenderPassCreateFlagBits`.

If the `VkRenderPassCreateInfo::pNext` chain includes a `VkRenderPassMultiviewCreateInfo` structure,
then that structure includes an array of view masks, view offsets, and correlation masks for the render pass.

The `VkRenderPassMultiviewCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkRenderPassMultiviewCreateInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           subpassCount;
    const uint32_t*    pViewMasks;
    uint32_t           dependencyCount;
    const int32_t*     pViewOffsets;
    uint32_t           correlationMaskCount;
    const uint32_t*    pCorrelationMasks;
} VkRenderPassMultiviewCreateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_multiview
typedef VkRenderPassMultiviewCreateInfo VkRenderPassMultiviewCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `subpassCount` is zero or the number of subpasses in the render pass.
- `pViewMasks` is a pointer to an array of `subpassCount` view masks, where each mask is a bitfield of view indices describing which views rendering is broadcast to in each subpass, when multiview is enabled. If `subpassCount` is zero, each view mask is treated as zero.
- `dependencyCount` is zero or the number of dependencies in the render pass.
- `pViewOffsets` is a pointer to an array of `dependencyCount` view offsets, one for each dependency. If `dependencyCount` is zero, each dependency's view offset is treated as zero. Each view offset controls which views in the source subpass the views in the destination subpass depend on.
- `correlationMaskCount` is zero or the number of correlation masks.
- `pCorrelationMasks` is a pointer to an array of `correlationMaskCount` view masks indicating sets of views that `may` be more efficient to render concurrently.

When a subpass uses a non-zero view mask, `multiview` functionality is considered to be enabled. Multiview is all-or-nothing for a render pass - that is, either all subpasses `must` have a non-zero view mask (though some subpasses `may` have only one view) or all `must` be zero. Multiview causes all drawing and clear commands in the subpass to behave as if they were broadcast to each view, where a view is represented by one layer of the framebuffer attachments. All draws and clears are broadcast to each `view index` whose bit is set in the view mask. The view index is provided in the `ViewIndex` shader input variable, and color, depth/stencil, and input attachments all read/write the layer of the framebuffer corresponding to the view index.
If the view mask is zero for all subpasses, multiview is considered to be disabled and all drawing commands execute normally, without this additional broadcasting.

Some implementations may not support multiview in conjunction with geometry shaders or tessellation shaders.

When multiview is enabled, the `VK_DEPENDENCY_VIEW_LOCAL_BIT` bit in a dependency can be used to express a view-local dependency, meaning that each view in the destination subpass depends on a single view in the source subpass. Unlike pipeline barriers, a subpass dependency can potentially have a different view mask in the source subpass and the destination subpass. If the dependency is view-local, then each view (dstView) in the destination subpass depends on the view dstView + pViewOffsets[dependency] in the source subpass. If there is not such a view in the source subpass, then this dependency does not affect that view in the destination subpass. If the dependency is not view-local, then all views in the destination subpass depend on all views in the source subpass, and the view offset is ignored. A non-zero view offset is not allowed in a self-dependency.

The elements of `pCorrelationMasks` are a set of masks of views indicating that views in the same mask may exhibit spatial coherency between the views, making it more efficient to render them concurrently. Correlation masks must not have a functional effect on the results of the multiview rendering.

When multiview is enabled, at the beginning of each subpass all non-render pass state is undefined. In particular, each time `vkCmdBeginRenderPass` or `vkCmdNextSubpass` is called the graphics pipeline must be bound, any relevant descriptor sets or vertex/index buffers must be bound, and any relevant dynamic state or push constants must be set before they are used.

A multiview subpass can declare that its shaders will write per-view attributes for all views in a single invocation, by setting the `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX` bit in the subpass description. The only supported per-view attributes are position and viewport mask, and per-view position and viewport masks are written to output array variables decorated with `PositionPerViewNV` and `ViewportMaskPerViewNV`, respectively. If `VK_NV_viewport_array2` is not supported and enabled, `ViewportMaskPerViewNV` must not be used. Values written to elements of `PositionPerViewNV` and `ViewportMaskPerViewNV` must not depend on the `ViewIndex`. The shader must also write to an output variable decorated with `Position`, and the value written to `Position` must equal the value written to `PositionPerViewNV[ViewIndex]`. Similarly, if `ViewportMaskPerViewNV` is written to then the shader must also write to an output variable decorated with `ViewportMaskNV`, and the value written to `ViewportMaskNV` must equal the value written to `ViewportMaskPerViewNV[ViewIndex]`. Implementations will either use values taken from `Position` and `ViewportMaskNV` and invoke the shader once for each view, or will use values taken from `PositionPerViewNV` and `ViewportMaskPerViewNV` and invoke the shader fewer times. The values written to `Position` and `ViewportMaskNV` must not depend on the values written to `PositionPerViewNV` and `ViewportMaskPerViewNV`, or vice versa (to allow compilers to eliminate the unused outputs). All attributes that do not have *PerViewNV counterparts must not depend on `ViewIndex`.

Per-view attributes are all-or-nothing for a subpass. That is, all pipelines compiled against a subpass that includes the `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX` bit must write per-view attributes to the *PerViewNV[] shader outputs, in addition to the non-per-view (e.g. `Position`) outputs. Pipelines compiled against a subpass that does not include this bit must not include the *PerViewNV[] outputs in their interfaces.
Valid Usage

- VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-00841
  Each view index must not be set in more than one element of pCorrelationMasks

Valid Usage (Implicit)

- VUID-VkRenderPassMultiviewCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO

- VUID-VkRenderPassMultiviewCreateInfo-pViewMasks-parameter
  If subpassCount is not 0, pViewMasks must be a valid pointer to an array of subpassCount uint32_t values

- VUID-VkRenderPassMultiviewCreateInfo-pViewOffsets-parameter
  If dependencyCount is not 0, pViewOffsets must be a valid pointer to an array of dependencyCount int32_t values

- VUID-VkRenderPassMultiviewCreateInfo-pCorrelationMasks-parameter
  If correlationMaskCount is not 0, pCorrelationMasks must be a valid pointer to an array of correlationMaskCount uint32_t values

If the VkRenderPassCreateInfo::pNext chain includes a VkRenderPassFragmentDensityMapCreateInfoEXT structure, then that structure includes a fragment density map attachment for the render pass.

The VkRenderPassFragmentDensityMapCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_fragment_density_map
typedef struct VkRenderPassFragmentDensityMapCreateInfoEXT {
  VkStructureType          sType;
  const void              *pNext;
  VkAttachmentReference    fragmentDensityMapAttachment;
} VkRenderPassFragmentDensityMapCreateInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- fragmentDensityMapAttachment is the fragment density map to use for the render pass.

The fragment density map is read at an implementation-dependent time with the following constraints determined by the attachment's image view flags:

- VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT specifies that the fragment density map will be read by the device during VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT
- VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DEFERRED_BIT_EXT specifies that the fragment density map will be read by the host during vkEndCommandBuffer of the primary command buffer that the render pass is recorded into
• Otherwise the fragment density map will be read by the host during `vkCmdBeginRenderPass`
The fragment density map may additionally be read by the device during `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT` for any mode.

If this structure is not present, it is as if `fragmentDensityMapAttachment` was given as `VK_ATTACHMENT_UNUSED`.

Valid Usage

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-fragmentDensityMapAttachment-02547
  If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` must be less than `VkRenderPassCreateInfo::attachmentCount`

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-fragmentDensityMapAttachment-02548
  If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` must not be an element of `VkSubpassDescription::pInputAttachments, VkSubpassDescription::pColorAttachments, VkSubpassDescription::pResolveAttachments, VkSubpassDescription::pDepthStencilAttachment, or VkSubpassDescription::pPreserveAttachments` for any subpass

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-fragmentDensityMapAttachment-02549
  If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, layout must be equal to `VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT`, or `VK_IMAGE_LAYOUT_GENERAL`

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-fragmentDensityMapAttachment-02550
  If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` must reference an attachment with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_LOAD` or `VK_ATTACHMENT_LOAD_OP_DONT_CARE`

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-fragmentDensityMapAttachment-02551
  If `fragmentDensityMapAttachment` is not `VK_ATTACHMENT_UNUSED`, `fragmentDensityMapAttachment` must reference an attachment with a `storeOp` equal to `VK_ATTACHMENT.Store_OP_DONT_CARE`

Valid Usage (Implicit)

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_RENDER_PASS_FRAGMENT_DENSITY_MAP_CREATE_INFO_EXT`

• VUID-VkRenderPassFragmentDensityMapCreateInfoEXT-fragmentDensityMapAttachment-parameter
  `fragmentDensityMapAttachment` must be a valid `VkAttachmentReference` structure

The `VkAttachmentDescription` structure is defined as:
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
    VkAttachmentLoadOp stencilLoadOp;
    VkAttachmentStoreOp stencilStoreOp;
    VkImageLayout initialLayout;
    VkImageLayout finalLayout;
} VkAttachmentDescription;

- **flags** is a bitmask of VkAttachmentDescriptionFlagBits specifying additional properties of the attachment.
- **format** is a VkFormat value specifying the format of the image view that will be used for the attachment.
- **samples** is the number of samples of the image as defined in VkSampleCountFlagBits.
- **loadOp** is a VkAttachmentLoadOp value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
- **storeOp** is a VkAttachmentStoreOp value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
- **stencilLoadOp** is a VkAttachmentLoadOp value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
- **stencilStoreOp** is a VkAttachmentStoreOp value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
- **initialLayout** is the layout the attachment image subresource will be in when a render pass instance begins.
- **finalLayout** is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

If the attachment uses a color format, then loadOp and storeOp are used, and stencilLoadOp and stencilStoreOp are ignored. If the format has depth and/or stencil components, loadOp and storeOp apply only to the depth data, while stencilLoadOp and stencilStoreOp define how the stencil data is handled. loadOp and stencilLoadOp define the load operations that execute as part of the first subpass that uses the attachment. storeOp and stencilStoreOp define the store operations that execute as part of the last subpass that uses the attachment.

The load operation for each sample in an attachment happens-before any recorded command which accesses the sample in the first subpass where the attachment is used. Load operations for attachments with a depth/stencil format execute in the VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT pipeline stage. Load operations for attachments with a color format execute in the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT pipeline stage.

The store operation for each sample in an attachment happens-after any recorded command which
accesses the sample in the last subpass where the attachment is used. Store operations for attachments with a depth/stencil format execute in the VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT pipeline stage. Store operations for attachments with a color format execute in the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT pipeline stage.

If an attachment is not used by any subpass, then loadOp, storeOp, stencilStoreOp, and stencilLoadOp are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

The load and store operations apply on the first and last use of each view in the render pass, respectively. If a view index of an attachment is not included in the view mask in any subpass that uses it, then the load and store operations are ignored, and the attachment's memory contents will not be modified by execution of a render pass instance.

During a render pass instance, input/color attachments with color formats that have a component size of 8, 16, or 32 bits must be represented in the attachment's format throughout the instance. Attachments with other floating- or fixed-point color formats, or with depth components may be represented in a format with a precision higher than the attachment format, but must be represented with the same range. When such a component is loaded via the loadOp, it will be converted into an implementation-dependent format used by the render pass. Such components must be converted from the render pass format, to the format of the attachment, before they are resolved or stored at the end of a render pass instance via storeOp. Conversions occur as described in Numeric Representation and Computation and Fixed-Point Data Conversions.

If flags includes VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, then the attachment is treated as if it shares physical memory with another attachment in the same render pass. This information limits the ability of the implementation to reorder certain operations (like layout transitions and the loadOp) such that it is not improperly reordered against other uses of the same physical memory via a different attachment. This is described in more detail below.

If a render pass uses multiple attachments that alias the same device memory, those attachments must each include the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit in their attachment description flags. Attachments aliasing the same memory occurs in multiple ways:

- Multiple attachments being assigned the same image view as part of framebuffer creation.
- Attachments using distinct image views that correspond to the same image subresource of an image.
- Attachments using views of distinct image subresources which are bound to overlapping memory ranges.

Note

Render passes must include subpass dependencies (either directly or via a subpass dependency chain) between any two subpasses that operate on the same attachment or aliasing attachments and those subpass dependencies must include execution and memory dependencies separating uses of the aliases, if at least one of those subpasses writes to one of the aliases. These dependencies must not include the VK_DEPENDENCY_BY_REGION_BIT if the aliases are views of distinct image subresources which overlap in memory.
Multiple attachments that alias the same memory **must** not be used in a single subpass. A given attachment index **must** not be used multiple times in a single subpass, with one exception: two subpass attachments **can** use the same attachment index if at least one use is as an input attachment and neither use is as a resolve or preserve attachment. In other words, the same view **can** be used simultaneously as an input and color or depth/stencil attachment, but **must** not be used as multiple color or depth/stencil attachments nor as resolve or preserve attachments. The precise set of valid scenarios is described in more detail below.

If a set of attachments alias each other, then all except the first to be used in the render pass **must** use an initialLayout of VK_IMAGE_LAYOUT_UNDEFINED, since the earlier uses of the other aliases make their contents undefined. Once an alias has been used and a different alias has been used after it, the first alias **must** not be used in any later subpasses. However, an application **can** assign the same image view to multiple aliasing attachment indices, which allows that image view to be used multiple times even if other aliases are used in between.

**Note**

Once an attachment needs the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit, there **should** be no additional cost of introducing additional aliases, and using these additional aliases **may** allow more efficient clearing of the attachments on multiple uses via VK_ATTACHMENT_LOAD_OP_CLEAR.
Valid Usage

- VUID-VkAttachmentDescription-finalLayout-00843
  `finalLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`.

- VUID-VkAttachmentDescription-format-03280
  If `format` is a color format, `initialLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

- VUID-VkAttachmentDescription-format-03281
  If `format` is a depth/stencil format, `initialLayout` must not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`.

- VUID-VkAttachmentDescription-format-03282
  If `format` is a color format, `finalLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

- VUID-VkAttachmentDescription-format-03283
  If `format` is a depth/stencil format, `finalLayout` must not be `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`.

- VUID-VkAttachmentDescription-separateDepthStencilLayouts-03284
  If the `separateDepthStencilLayouts` feature is not enabled, `initialLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- VUID-VkAttachmentDescription-separateDepthStencilLayouts-03285
  If the `separateDepthStencilLayouts` feature is not enabled, `finalLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- VUID-VkAttachmentDescription-format-03286
  If `format` is a color format, `initialLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- VUID-VkAttachmentDescription-format-03287
  If `format` is a color format, `finalLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- VUID-VkAttachmentDescription-format-03288
  If `format` is a depth/stencil format which includes both depth and stencil aspects, `initialLayout` must not be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.
If format is a depth/stencil format which includes both depth and stencil aspects, finalLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a depth/stencil format which includes only the depth aspect, initialLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a depth/stencil format which includes only the depth aspect, finalLayout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

If format is a depth/stencil format which includes only the stencil aspect, initialLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL.

If format is a depth/stencil format which includes only the stencil aspect, finalLayout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL.
Valid Usage (Implicit)

- **flags** must be a valid combination of VkAttachmentDescriptionFlagBits values.

- **format** must be a valid VkFormat value.

- **samples** must be a valid VkSampleCountFlagBits value.

- **loadOp** must be a valid VkAttachmentLoadOp value.

- **storeOp** must be a valid VkAttachmentStoreOp value.

- **stencilLoadOp** must be a valid VkAttachmentLoadOp value.

- **stencilStoreOp** must be a valid VkAttachmentStoreOp value.

- **initialLayout** must be a valid VkImageLayout value.

- **finalLayout** must be a valid VkImageLayout value.

Bits which can be set in VkAttachmentDescription::flags describing additional properties of the attachment are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkAttachmentDescriptionFlagBits {
    VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT = 0x00000001,
} VkAttachmentDescriptionFlagBits;
```

- **VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT** specifies that the attachment aliases the same device memory as other attachments.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkAttachmentDescriptionFlags;
```

VkAttachmentDescriptionFlags is a bitmask type for setting a mask of zero or more VkAttachmentDescriptionFlagBits.

Possible values of VkAttachmentDescription::loadOp and stencilLoadOp, specifying how the contents of the attachment are treated, are:
typedef enum VkAttachmentLoadOp {
    VK_ATTACHMENT_LOAD_OP_LOAD = 0,
    VK_ATTACHMENT_LOAD_OP_CLEAR = 1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE = 2,
} VkAttachmentLoadOp;

- **VK_ATTACHMENT_LOAD_OP_LOAD** specifies that the previous contents of the image within the render area will be preserved. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`.

- **VK_ATTACHMENT_LOAD_OP_CLEAR** specifies that the contents within the render area will be cleared to a uniform value, which is specified when a render pass instance is begun. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

- **VK_ATTACHMENT_LOAD_OP_DONT_CARE** specifies that the previous contents within the area need not be preserved; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

Possible values of `VkAttachmentDescription::storeOp` and `stencilStoreOp`, specifying how the contents of the attachment are treated, are:

typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
    // Provided by VK_QCOM_render_pass_store_ops
    VK_ATTACHMENT_STORE_OP_NONE_QCOM = 1000301000,
} VkAttachmentStoreOp;

- **VK_ATTACHMENT_STORE_OP_STORE** specifies the contents generated during the render pass and within the render area are written to memory. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

- **VK_ATTACHMENT_STORE_OP_DONT_CARE** specifies the contents within the render area are not needed after rendering, and **may** be discarded; the contents of the attachment will be undefined inside the render area. For attachments with a depth/stencil format, this uses the access type `VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT`. For attachments with a color format, this uses the access type `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`.

- **VK_ATTACHMENT_STORE_OP_NONE_QCOM** specifies that the contents within the render area were not written during rendering, and **may** not be written to memory. If the attachment was written to during the renderpass, the contents of the attachment will be undefined inside the render area.
Note

`VK_ATTACHMENT_STORE_OP_DONT_CARE` can cause contents generated during previous render passes to be discarded before reaching memory, even if no write to the attachment occurs during the current render pass.

To specify which aspects of an input attachment can be read, add a `VkRenderPassInputAttachmentAspectCreateInfo` structure to the `pNext` chain of the `VkRenderPassCreateInfo` structure:

The `VkRenderPassInputAttachmentAspectCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkRenderPassInputAttachmentAspectCreateInfo {
    VkStructureType                           sType;
    const void*                                pNext;
    uint32_t                                    aspectReferenceCount;
    const VkInputAttachmentAspectReference*    pAspectReferences;
} VkRenderPassInputAttachmentAspectCreateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance2
typedef VkRenderPassInputAttachmentAspectCreateInfo
VkRenderPassInputAttachmentAspectCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `aspectReferenceCount` is the number of elements in the `pAspectReferences` array.
- `pAspectReferences` is a pointer to an array of `aspectReferenceCount` `VkInputAttachmentAspectReference` structures containing a mask describing which aspect(s) can be accessed for a given input attachment within a given subpass.

An application can access any aspect of an input attachment that does not have a specified aspect mask in the `pAspectReferences` array. Otherwise, an application must not access aspect(s) of an input attachment other than those in its specified aspect mask.
Valid Usage (Implicit)

- **VUID-VkRenderPassInputAttachmentAspectCreateInfo-sType-sType**
  
  sType must be `VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO`

- **VUID-VkRenderPassInputAttachmentAspectCreateInfo-pAspectReferences-parameter**
  
  pAspectReferences must be a valid pointer to an array of `aspectReferenceCount` valid `VkInputAttachmentAspectReference` structures

- **VUID-VkRenderPassInputAttachmentAspectCreateInfo-aspectReferenceCount-arraylength**
  
  `aspectReferenceCount` must be greater than 0

The `VkInputAttachmentAspectReference` structure specifies an aspect mask for a specific input attachment of a specific subpass in the render pass.

*subpass* and *inputAttachmentIndex* index into the render pass as:

```
pCreateInfo->pSubpasses[subpass].pInputAttachments[inputAttachmentIndex]
```

// Provided by VK_VERSION_1_1
```c
typedef struct VkInputAttachmentAspectReference {
    uint32_t subpass;
    uint32_t inputAttachmentIndex;
    VkImageAspectFlags aspectMask;
} VkInputAttachmentAspectReference;
```

// Provided by VK_KHR_maintenance2
```c
typedef VkInputAttachmentAspectReference VkInputAttachmentAspectReferenceKHR;
```

- subpass is an index into the *pSubpasses* array of the parent `VkRenderPassCreateInfo` structure.
- inputAttachmentIndex is an index into the *pInputAttachments* of the specified subpass.
- aspectMask is a mask of which aspect(s) can be accessed within the specified subpass.

Valid Usage

- **VUID-VkInputAttachmentAspectReference-aspectMask-01964**
  
  aspectMask must not include `VK_IMAGE_ASPECT_METADATA_BIT`

- **VUID-VkInputAttachmentAspectReference-aspectMask-02250**
  
  aspectMask must not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index i
Valid Usage (Implicit)

- VUID-VkInputAttachmentAspectReference-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values

- VUID-VkInputAttachmentAspectReference-aspectMask-requiredbitmask
  aspectMask must not be 0

The VkSubpassDescription structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags       flags;
    VkPipelineBindPoint             pipelineBindPoint;
    uint32_t                        inputAttachmentCount;
    const VkAttachmentReference*    pInputAttachments;
    uint32_t                        colorAttachmentCount;
    const VkAttachmentReference*    pColorAttachments;
    const VkAttachmentReference*    pResolveAttachments;
    const VkAttachmentReference*    pDepthStencilAttachment;
    uint32_t                        preserveAttachmentCount;
    const uint32_t*                 pPreserveAttachments;
} VkSubpassDescription;
```

- **flags** is a bitmask of VkSubpassDescriptionFlagBits specifying usage of the subpass.
- **pipelineBindPoint** is a VkPipelineBindPoint value specifying the pipeline type supported for this subpass.
- **inputAttachmentCount** is the number of input attachments.
- **pInputAttachments** is a pointer to an array of VkAttachmentReference structures defining the input attachments for this subpass and their layouts.
- **colorAttachmentCount** is the number of color attachments.
- **pColorAttachments** is a pointer to an array of VkAttachmentReference structures defining the color attachments for this subpass and their layouts.
- **pResolveAttachments** is an optional array of colorAttachmentCount VkAttachmentReference structures defining the resolve attachments for this subpass and their layouts.
- **pDepthStencilAttachment** is a pointer to a VkAttachmentReference structure specifying the depth/stencil attachment for this subpass and its layout.
- **preserveAttachmentCount** is the number of preserved attachments.
- **pPreserveAttachments** is a pointer to an array of preserveAttachmentCount render pass attachment indices identifying attachments that are not used by this subpass, but whose contents must be preserved throughout the subpass.

Each element of the pInputAttachments array corresponds to an input attachment index in a fragment shader, i.e. if a shader declares an image variable decorated with a InputAttachmentIndex
value of $X$, then it uses the attachment provided in $\text{pInputAttachments}[X]$. Input attachments must also be bound to the pipeline in a descriptor set. If the attachment member of any element of $\text{pInputAttachments}$ is $\text{VK_ATTACHMENT_UNUSED}$, the application must not read from the corresponding input attachment index. Fragment shaders can use subpass input variables to access the contents of an input attachment at the fragment's $(x, y, \text{layer})$ framebuffer coordinates. Input attachments must not be used by any subpasses within a renderpass that enables render pass transform.

Each element of the $\text{pColorAttachments}$ array corresponds to an output location in the shader, i.e. if the shader declares an output variable decorated with a Location value of $X$, then it uses the attachment provided in $\text{pColorAttachments}[X]$. If the attachment member of any element of $\text{pColorAttachments}$ is $\text{VK_ATTACHMENT_UNUSED}$, or if Color Write Enable has been disabled for the corresponding attachment index, then writes to the corresponding location by a fragment are discarded.

If flags does not include $\text{VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM}$, and if $\text{pResolveAttachments}$ is not NULL, each of its elements corresponds to a color attachment (the element in $\text{pColorAttachments}$ at the same index), and a multisample resolve operation is defined for each attachment. At the end of each subpass, multisample resolve operations read the subpass's color attachments, and resolve the samples for each pixel within the render area to the same pixel location in the corresponding resolve attachments, unless the resolve attachment index is $\text{VK_ATTACHMENT_UNUSED}$.

Similarly, if flags does not include $\text{VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM}$, and $\text{VkSubpassDescriptionDepthStencilResolve}::\text{pDepthStencilResolveAttachment}$ is not NULL and does not have the value $\text{VK_ATTACHMENT_UNUSED}$, it corresponds to the depth/stencil attachment in $\text{pDepthStencilAttachment}$, and multisample resolve operations for depth and stencil are defined by $\text{VkSubpassDescriptionDepthStencilResolve}::\text{depthResolveMode}$ and $\text{VkSubpassDescriptionDepthStencilResolve}::\text{stencilResolveMode}$, respectively. At the end of each subpass, multisample resolve operations read the subpass's depth/stencil attachment, and resolve the samples for each pixel to the same pixel location in the corresponding resolve attachment. If $\text{VkSubpassDescriptionDepthStencilResolve}::\text{depthResolveMode}$ is $\text{VK_RESOLVE_MODE_NONE}$, then the depth component of the resolve attachment is not written to and its contents are preserved. Similarly, if $\text{VkSubpassDescriptionDepthStencilResolve}::\text{stencilResolveMode}$ is $\text{VK_RESOLVE_MODE_NONE}$, then the stencil component of the resolve attachment is not written to and its contents are preserved. $\text{VkSubpassDescriptionDepthStencilResolve}::\text{depthResolveMode}$ is ignored if the $\text{VkFormat}$ of the $\text{pDepthStencilResolveAttachment}$ does not have a depth component. Similarly, $\text{VkSubpassDescriptionDepthStencilResolve}::\text{stencilResolveMode}$ is ignored if the $\text{VkFormat}$ of the $\text{pDepthStencilResolveAttachment}$ does not have a stencil component.

If the image subresource range referenced by the depth/stencil attachment is created with $\text{VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT}$, then the multisample resolve operation uses the sample locations state specified in the sampleLocationsInfo member of the element of the $\text{VkRenderPassSampleLocationsBeginInfoEXT}::\text{pPostSubpassSampleLocations}$ for the subpass.

If $\text{pDepthStencilAttachment}$ is NULL, or if its attachment index is $\text{VK_ATTACHMENT_UNUSED}$, it indicates that no depth/stencil attachment will be used in the subpass.

The contents of an attachment within the render area become undefined at the start of a subpass S
if all of the following conditions are true:

- The attachment is used as a color, depth/stencil, or resolve attachment in any subpass in the render pass.
- There is a subpass $S_1$ that uses or preserves the attachment, and a subpass dependency from $S_1$ to $S$.
- The attachment is not used or preserved in subpass $S$.

In addition, the contents of an attachment within the render area become undefined at the start of a subpass $S$ if all of the following conditions are true:

- $\text{VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM}$ is set.
- The attachment is used as a color or depth/stencil in the subpass.

Once the contents of an attachment become undefined in subpass $S$, they remain undefined for subpasses in subpass dependency chains starting with subpass $S$ until they are written again. However, they remain valid for subpasses in other subpass dependency chains starting with subpass $S$, if those subpasses use or preserve the attachment.
Valid Usage

- VUID-VkSubpassDescription-pipelineBindPoint-00844
  pipelineBindPoint must be VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-VkSubpassDescription-colorAttachmentCount-00845
  colorAttachmentCount must be less than or equal to VkPhysicalDeviceLimits::maxColorAttachments

- VUID-VkSubpassDescription-loadOp-00846
  If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then loadOp must not be VK_ATTACHMENT_LOAD_OP_CLEAR

- VUID-VkSubpassDescription-pResolveAttachments-00847
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not be VK_ATTACHMENT_UNUSED

- VUID-VkSubpassDescription-pResolveAttachments-00848
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT

- VUID-VkSubpassDescription-pResolveAttachments-00849
  If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have a sample count of VK_SAMPLE_COUNT_1_BIT

- VUID-VkSubpassDescription-pColorAttachments-01417
  All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have the same sample count

- VUID-VkSubpassDescription-pInputAttachments-02647
  All attachments in pInputAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain at least VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT or VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

- VUID-VkSubpassDescription-pColorAttachments-02648
  All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

- VUID-VkSubpassDescription-pResolveAttachments-02649
  All attachments in pResolveAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

- VUID-VkSubpassDescription-pDepthStencilAttachment-02650
  If pDepthStencilAttachment is not NULL and the attachment is not VK_ATTACHMENT_UNUSED then it must have an image format whose potential format features contain
If the `VK_AMD_mixed_attachment_samples` extension is enabled, and all attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` must have a sample count that is smaller than or equal to the sample count of `pDepthStencilAttachment` if it is not `VK_ATTACHMENT_UNUSED`.

If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `pDepthStencilAttachment` is not `VK_ATTACHMENT_UNUSED` and any attachments in `pColorAttachments` are not `VK_ATTACHMENT_UNUSED`, they must have the same sample count.

The attachment member of each element of `pPreserveAttachments` must not be `VK_ATTACHMENT_UNUSED`.

Each element of `pPreserveAttachments` must not also be an element of any other member of the subpass description.

If any attachment is used by more than one `VkAttachmentReference` member, then each use must use the same layout.

Each attachment must follow the image layout requirements specified for its attachment type.

If `flags` includes `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX`, it must also include `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX`.

If `flags` includes `VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM`, and if `pResolveAttachments` is not `NULL`, then each resolve attachment must be `VK_ATTACHMENT_UNUSED`.

If `flags` includes `VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM`, and if `pDepthStencilResolveAttachmentKHR` is not `NULL`, then the depth/stencil resolve attachment must be `VK_ATTACHMENT_UNUSED`.

If `flags` includes `VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM`, then the subpass must be the last subpass in a subpass dependency chain.

If `flags` includes `VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM`, then the sample count of the input attachments must equal `rasterizationSamples`.

If `flags` includes `VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM`, and if `sampleShadingEnable` is enabled (explicitly or implicitly) then `minSampleShading` must equal 0.0.
If the render pass is created with `VK_RENDER_PASS_CREATE_TRANSFORM_BIT_QCOM` each of the elements of `pInputAttachments` must be `VK_ATTACHMENT_UNUSED`.

`pDepthStencilAttachment` and `pColorAttachments` must not contain references to the same attachment.

### Valid Usage (Implicit)

- **flags** must be a valid combination of `VkSubpassDescriptionFlagBits` values.
- `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value.
- If `inputAttachmentCount` is not 0, `pInputAttachments` must be a valid pointer to an array of `inputAttachmentCount` valid `VkAttachmentReference` structures.
- If `colorAttachmentCount` is not 0, `pColorAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures.
- If `colorAttachmentCount` is not 0, and `pResolveAttachments` is not NULL, `pResolveAttachments` must be a valid pointer to an array of `colorAttachmentCount` valid `VkAttachmentReference` structures.
- If `pDepthStencilAttachment` is not NULL, `pDepthStencilAttachment` must be a valid pointer to a valid `VkAttachmentReference` structure.
- If `preserveAttachmentCount` is not 0, `pPreserveAttachments` must be a valid pointer to an array of `preserveAttachmentCount` `uint32_t` values.

Bits which can be set in `VkSubpassDescription::flags`, specifying usage of the subpass, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSubpassDescriptionFlagBits {
    // Provided by VK_NVX_multiview_per_view_attributes
    VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX = 0x00000001,
    // Provided by VK_NVX_multiview_per_view_attributes
    VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX = 0x00000002,
    // Provided by VK_QCOM_render_pass_shader_resolve
    VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM = 0x00000004,
    // Provided by VK_QCOM_render_pass_shader_resolve
    VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM = 0x00000008,
} VkSubpassDescriptionFlagBits;
```
• **VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX** specifies that shaders compiled for this subpass write the attributes for all views in a single invocation of each vertex processing stage. All pipelines compiled against a subpass that includes this bit **must** write per-view attributes to the *PerViewNV[]* shader outputs, in addition to the non-per-view (e.g. *Position*) outputs.

• **VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX** specifies that shaders compiled for this subpass use per-view positions which only differ in value in the x component. Per-view viewport mask **can** also be used.

• **VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM** specifies that the framebuffer region is the fragment region, that is, the minimum region dependencies are by pixel rather than by sample, such that any fragment shader invocation **can** access any sample associated with that fragment shader invocation.

• **VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM** specifies that the subpass performs shader resolve operations.

  \[\text{Note}\]
  Shader resolve operations allow for custom resolve operations, but overwriting pixels **may** have a performance and/or power cost. Furthermore, since the content of any depth stencil attachment or color attachment is undefined at the begining of a shader resolve subpass, any depth testing, stencil testing, or blending operation which sources these undefined values also has undefined result value.

  // Provided by VK_VERSION_1_0
  typedef VkFlags VkSubpassDescriptionFlags;

  **VkSubpassDescriptionFlags** is a bitmask type for setting a mask of zero or more **VkSubpassDescriptionFlagBits**.

The **VkAttachmentReference** structure is defined as:

  // Provided by VK_VERSION_1_0
  typedef struct VkAttachmentReference {
    uint32_t attachment;
    VkImageLayout layout;
  } VkAttachmentReference;

• **attachment** is either an integer value identifying an attachment at the corresponding index in **VkRenderPassCreateInfo::pAttachments**, or **VK_ATTACHMENT_UNUSED** to signify that this attachment is not used.

• **layout** is a **VkImageLayout** value specifying the layout the attachment uses during the subpass.
Valid Usage

- **VUID-VkAttachmentReference-layout-00857**

  If `attachment` is not `VK_ATTACHMENT_UNUSED`, layout **must** not be `VK_IMAGE_LAYOUT_UNDEFINED`, `VK_IMAGE_LAYOUT_PREINITIALIZED`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

Valid Usage (Implicit)

- **VUID-VkAttachmentReference-layout-parameter**

  layout **must** be a valid `VkImageLayout` value

The `VkSubpassDependency` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSubpassDependency {
    uint32_t                srcSubpass;
    uint32_t                dstSubpass;
    VkPipelineStageFlags    srcStageMask;
    VkPipelineStageFlags    dstStageMask;
    VkAccessFlags           srcAccessMask;
    VkAccessFlags           dstAccessMask;
    VkDependencyFlags       dependencyFlags;
} VkSubpassDependency;
```

- `srcSubpass` is the subpass index of the first subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- `dstSubpass` is the subpass index of the second subpass in the dependency, or `VK_SUBPASS_EXTERNAL`.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the source stage mask.
- `dstStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying the destination stage mask.
- `srcAccessMask` is a bitmask of `VkAccessFlagBits` specifying a source access mask.
- `dstAccessMask` is a bitmask of `VkAccessFlagBits` specifying a destination access mask.
- `dependencyFlags` is a bitmask of `VkDependencyFlagBits`.

If `srcSubpass` is equal to `dstSubpass` then the `VkSubpassDependency` describes a **subpass self-dependency**, and only constrains the pipeline barriers allowed within a subpass instance. Otherwise, when a render pass instance which includes a subpass dependency is submitted to a queue, it defines a memory dependency between the subpasses identified by `srcSubpass` and `dstSubpass`.

If `srcSubpass` is equal to `VK_SUBPASS_EXTERNAL`, the first **synchronization scope** includes commands that occur earlier in submission order than the `vkCmdBeginRenderPass` used to begin the render.
pass instance. Otherwise, the first set of commands includes all commands submitted as part of the subpass instance identified by srcSubpass and any load, store or multisample resolve operations on attachments used in srcSubpass. In either case, the first synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by srcStageMask.

If dstSubpass is equal to VK_SUBPASS_EXTERNAL, the second synchronization scope includes commands that occur later in submission order than the vkCmdEndRenderPass used to end the render pass instance. Otherwise, the second set of commands includes all commands submitted as part of the subpass instance identified by dstSubpass and any load, store or multisample resolve operations on attachments used in dstSubpass. In either case, the second synchronization scope is limited to operations on the pipeline stages determined by the destination stage mask specified by dstStageMask.

The first access scope is limited to access in the pipeline stages determined by the source stage mask specified by srcStageMask. It is also limited to access types in the source access mask specified by srcAccessMask.

The second access scope is limited to access in the pipeline stages determined by the destination stage mask specified by dstStageMask. It is also limited to access types in the destination access mask specified by dstAccessMask.

The availability and visibility operations defined by a subpass dependency affect the execution of image layout transitions within the render pass.

**Note**

For non-attachment resources, the memory dependency expressed by subpass dependency is nearly identical to that of a VkMemoryBarrier (with matching srcAccessMask and dstAccessMask parameters) submitted as a part of a vkCmdPipelineBarrier (with matching srcStageMask and dstStageMask parameters). The only difference being that its scopes are limited to the identified subpasses rather than potentially affecting everything before and after.

For attachments however, subpass dependencies work more like a VkImageMemoryBarrier defined similarly to the VkMemoryBarrier above, the queue family indices set to VK_QUEUE_FAMILY_IGNORED, and layouts as follows:

- The equivalent to oldLayout is the attachment's layout according to the subpass description for srcSubpass.
- The equivalent to newLayout is the attachment's layout according to the subpass description for dstSubpass.
Valid Usage

- **VUID-VkSubpassDependency-srcStageMask-00860**
  If the geometry shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

- **VUID-VkSubpassDependency-dstStageMask-00861**
  If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

- **VUID-VkSubpassDependency-srcStageMask-00862**
  If the tessellation shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-VkSubpassDependency-dstStageMask-00863**
  If the tessellation shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`.

- **VUID-VkSubpassDependency-srcSubpass-00864**
  `srcSubpass` must be less than or equal to `dstSubpass`, unless one of them is `VK_SUBPASS_EXTERNAL`, to avoid cyclic dependencies and ensure a valid execution order.

- **VUID-VkSubpassDependency-srcSubpass-00865**
  `srcSubpass` and `dstSubpass` must not both be equal to `VK_SUBPASS_EXTERNAL`.

- **VUID-VkSubpassDependency-srcSubpass-00867**
  If `srcSubpass` is equal to `dstSubpass` and not all of the stages in `srcStageMask` and `dstStageMask` are framebuffer-space stages, the logically latest pipeline stage in `srcStageMask` must be logically earlier than or equal to the logically earliest pipeline stage in `dstStageMask`.

- **VUID-VkSubpassDependency-srcAccessMask-00868**
  Any access flag included in `srcAccessMask` must be supported by one of the pipeline stages in `srcStageMask`, as specified in the table of supported access types.

- **VUID-VkSubpassDependency-dstAccessMask-00869**
  Any access flag included in `dstAccessMask` must be supported by one of the pipeline stages in `dstStageMask`, as specified in the table of supported access types.

- **VUID-VkSubpassDependency-srcSubpass-02243**
  If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a framebuffer-space stage, then `dependencyFlags` must include `VK_DEPENDENCY_BY_REGION_BIT`.

- **VUID-VkSubpassDependency-dependencyFlags-02520**
  If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `srcSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`.

- **VUID-VkSubpassDependency-dependencyFlags-02521**
  If `dependencyFlags` includes `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `dstSubpass` must not be equal to `VK_SUBPASS_EXTERNAL`.

- **VUID-VkSubpassDependency-srcSubpass-00872**
  If `srcSubpass` equals `dstSubpass` and that subpass has more than one bit set in the view mask, then `dependencyFlags` must include `VK_DEPENDENCY_VIEW_LOCAL_BIT`. 
If the mesh shaders feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV

If the task shaders feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV

If the mesh shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV

If the task shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV

Valid Usage (Implicit)

srcStageMask must be a valid combination of VkPipelineStageFlagBits values

dstStageMask must be a valid combination of VkPipelineStageFlagBits values

srcAccessMask must be a valid combination of VkAccessFlagBits values

dstAccessMask must be a valid combination of VkAccessFlagBits values

dependencyFlags must be a valid combination of VkDependencyFlagBits values

When multiview is enabled, the execution of the multiple views of one subpass may not occur simultaneously or even back-to-back, and rather may be interleaved with the execution of other subpasses. The load and store operations apply to attachments on a per-view basis. For example, an attachment using VK_ATTACHMENT_LOAD_OP_CLEAR will have each view cleared on first use, but the first use of one view may be temporally distant from the first use of another view.
Note

A good mental model for multiview is to think of a multiview subpass as if it were a collection of individual (per-view) subpasses that are logically grouped together and described as a single multiview subpass in the API. Similarly, a multiview attachment can be thought of like several individual attachments that happen to be layers in a single image. A view-local dependency between two multiview subpasses acts like a set of one-to-one dependencies between corresponding pairs of per-view subpasses. A view-global dependency between two multiview subpasses acts like a set of $N \times M$ dependencies between all pairs of per-view subpasses in the source and destination. Thus, it is a more compact representation which also makes clear the commonality and reuse that is present between views in a subpass. This interpretation motivates the answers to questions like “when does the load op apply” - it is on the first use of each view of an attachment, as if each view were a separate attachment.

If any two subpasses of a render pass activate transform feedback to the same bound transform feedback buffers, a subpass dependency must be included (either directly or via some intermediate subpasses) between them.

If there is no subpass dependency from VK_SUBPASS_EXTERNAL to the first subpass that uses an attachment, then an implicit subpass dependency exists from VK_SUBPASS_EXTERNAL to the first subpass it is used in. The implicit subpass dependency only exists if there exists an automatic layout transition away from initialLayout. The subpass dependency operates as if defined with the following parameters:

```cpp
VkSubpassDependency implicitDependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL;
    .dstSubpass = firstSubpass; // First subpass attachment is used in
    .srcStageMask = VK_PIPELINE_STAGE_NONE_KHR;
    .dstStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .srcAccessMask = 0;
    .dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
        VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
        VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
        VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
        VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dependencyFlags = 0;
};
```

Similarly, if there is no subpass dependency from the last subpass that uses an attachment to VK_SUBPASS_EXTERNAL, then an implicit subpass dependency exists from the last subpass it is used in to VK_SUBPASS_EXTERNAL. The implicit subpass dependency only exists if there exists an automatic layout transition into finalLayout. The subpass dependency operates as if defined with the following parameters:
VkSubpassDependency implicitDependency = {
    .srcSubpass = lastSubpass; // Last subpass attachment is used in
    .dstSubpass = VK_SUBPASS_EXTERNAL;
    .srcStageMask = VK_PIPELINE_STAGE_ALL_COMMANDS_BIT;
    .dstStageMask = VK_PIPELINE_STAGE_NONE_KHR;
    .srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                     VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    .dstAccessMask = 0;
    .dependencyFlags = 0;
};

As subpasses may overlap or execute out of order with regards to other subpasses unless a subpass dependency chain describes otherwise, the layout transitions required between subpasses cannot be known to an application. Instead, an application provides the layout that each attachment must be in at the start and end of a render pass, and the layout it must be in during each subpass it is used in. The implementation then must execute layout transitions between subpasses in order to guarantee that the images are in the layouts required by each subpass, and in the final layout at the end of the render pass.

Automatic layout transitions apply to the entire image subresource attached to the framebuffer. If the attachment view is a 2D or 2D array view of a 3D image, even if the attachment view only refers to a subset of the slices of the selected mip level of the 3D image, automatic layout transitions apply to the entire subresource referenced which is the entire mip level in this case.

Automatic layout transitions away from the layout used in a subpass happen-after the availability operations for all dependencies with that subpass as the srcSubpass.

Automatic layout transitions into the layout used in a subpass happen-before the visibility operations for all dependencies with that subpass as the dstSubpass.

Automatic layout transitions away from initialLayout happens-after the availability operations for all dependencies with a srcSubpass equal to VK_SUBPASS_EXTERNAL, where dstSubpass uses the attachment that will be transitioned. For attachments created with VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, automatic layout transitions away from initialLayout happen-after the availability operations for all dependencies with a srcSubpass equal to VK_SUBPASS_EXTERNAL, where dstSubpass uses any aliased attachment.

Automatic layout transitions into finalLayout happens-before the visibility operations for all dependencies with a dstSubpass equal to VK_SUBPASS_EXTERNAL, where srcSubpass uses the attachment that will be transitioned. For attachments created with VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, automatic layout transitions into finalLayout happen-before the visibility operations for all dependencies with a dstSubpass equal to VK_SUBPASS_EXTERNAL, where srcSubpass uses any aliased attachment.

The image layout of the depth aspect of a depth/stencil attachment referring to an image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT is dependent on the last sample locations used to render to the attachment, thus automatic layout transitions use the sample locations state specified in VkRenderPassSampleLocationsBeginInfoEXT.
Automatic layout transitions of an attachment referring to a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` use the sample locations the image subresource range referenced by the attachment was last rendered with. If the current render pass does not use the attachment as a depth/stencil attachment in any subpass that happens-before, the automatic layout transition uses the sample locations state specified in the `sampleLocationsInfo` member of the element of the `VkRenderPassSampleLocationsBeginInfoEXT::pAttachmentInitialSampleLocations` array for which the `attachmentIndex` member equals the attachment index of the attachment, if one is specified. Otherwise, the automatic layout transition uses the sample locations state specified in the `sampleLocationsInfo` member of the element of the `VkRenderPassSampleLocationsBeginInfoEXT::pPostSubpassSampleLocations` array for which the `subpassIndex` member equals the index of the subpass that last used the attachment as a depth/stencil attachment, if one is specified.

If no sample locations state has been specified for an automatic layout transition performed on an attachment referring to a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` the contents of the depth aspect of the depth/stencil attachment become undefined as if the layout of the attachment was transitioned from the `VK_IMAGE_LAYOUT_UNDEFINED` layout.

If two subpasses use the same attachment, and both subpasses use the attachment in a read-only layout, no subpass dependency needs to be specified between those subpasses. If an implementation treats those layouts separately, it **must** insert an implicit subpass dependency between those subpasses to separate the uses in each layout. The subpass dependency operates as if defined with the following parameters:

```c
// Used for input attachments
VkPipelineStageFlags inputAttachmentStages = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
VkAccessFlags inputAttachmentDstAccess = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT;

// Used for depth/stencil attachments
VkPipelineStageFlags depthStencilAttachmentStages =
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
VkAccessFlags depthStencilAttachmentDstAccess =
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT;

VkSubpassDependency implicitDependency = {
    .srcSubpass = firstSubpass;
    .dstSubpass = secondSubpass;
    .srcStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .dstStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .srcAccessMask = 0;
    .dstAccessMask = inputAttachmentDstAccess | depthStencilAttachmentDstAccess;
    .dependencyFlags = 0;
};
```

If a subpass uses the same attachment as both an input attachment and either a color attachment or a depth/stencil attachment, writes via the color or depth/stencil attachment are not automatically...
made visible to reads via the input attachment, causing a feedback loop, except in any of the following conditions:

- If the color components or depth/stencil components read by the input attachment are mutually exclusive with the components written by the color or depth/stencil attachments, then there is no feedback loop. This requires the graphics pipelines used by the subpass to disable writes to color components that are read as inputs via the colorWriteEnable or colorWriteMask, and to disable writes to depth/stencil components that are read as inputs via depthWriteEnable or stencilTestEnable.

- If the attachment is used as an input attachment and depth/stencil attachment only, and the depth/stencil attachment is not written to.

Rendering within a subpass that contains a feedback loop creates a data race, except in the following cases:

- If a memory dependency is inserted between when the attachment is written and when it is subsequently read by later fragments. Pipeline barriers expressing a subpass self-dependency are the only way to achieve this, and one must be inserted every time a fragment will read values at a particular sample (x, y, layer, sample) coordinate, if those values have been written since the most recent pipeline barrier; or the since start of the subpass if there have been no pipeline barriers since the start of the subpass.

Attachments have requirements for a valid image layout depending on the usage

- An attachment used as an input attachment must be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout.

- An attachment used only as a color attachment must be in the VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR or VK_IMAGE_LAYOUT_GENERAL layout.

- An attachment used as both an input attachment and a color attachment must be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR or VK_IMAGE_LAYOUT_GENERAL layout.

- An attachment used only as a depth/stencil attachment must be in the VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout.

- An attachment used as an input attachment and depth/stencil attachment must be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout.
An attachment **must** not be used as both a depth/stencil attachment and a color attachment.

A more extensible version of render pass creation is also defined below.

To create a render pass, call:

```cpp
// Provided by VK_KHR_create_renderpass2
VkResult vkCreateRenderPass2KHR(
    VkDevice                                    device,
    const VkRenderPassCreateInfo2*              pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkRenderPass*                               pRenderPass);
```

- **device** is the logical device that creates the render pass.
- **pCreateInfo** is a pointer to a `VkRenderPassCreateInfo2` structure describing the parameters of the render pass.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pRenderPass** is a pointer to a `VkRenderPass` handle in which the resulting render pass object is returned.

This command is functionally identical to `vkCreateRenderPass`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage (Implicit)

- **VUID-vkCreateRenderPass2-device-parameter**
  
  * **device** must be a valid `VkDevice` handle

- **VUID-vkCreateRenderPass2-pCreateInfo-parameter**
  
  * **pCreateInfo** must be a valid pointer to a valid `VkRenderPassCreateInfo2` structure

- **VUID-vkCreateRenderPass2-pAllocator-parameter**
  
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateRenderPass2-pRenderPass-parameter**
  
  * **pRenderPass** must be a valid pointer to a `VkRenderPass` handle

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
The `VkRenderPassCreateInfo2` structure is defined as:

```c
typedef struct VkRenderPassCreateInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkRenderPassCreateFlags flags;
    uint32_t attachmentCount;
    const VkAttachmentDescription2* pAttachments;
    uint32_t subpassCount;
    const VkSubpassDescription2* pSubpasses;
    uint32_t dependencyCount;
    const VkSubpassDependency2* pDependencies;
    uint32_t correlatedViewMaskCount;
    const uint32_t* pCorrelatedViewMasks;
} VkRenderPassCreateInfo2;
```

or the equivalent

```c
// Provided by VK_KHR_create_renderpass2
typedef VkRenderPassCreateInfo2 VkRenderPassCreateInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `attachmentCount` is the number of attachments used by this render pass.
- `pAttachments` is a pointer to an array of `attachmentCount` `VkAttachmentDescription2` structures describing the attachments used by the render pass.
- `subpassCount` is the number of subpasses to create.
- `pSubpasses` is a pointer to an array of `subpassCount` `VkSubpassDescription2` structures describing each subpass.
- `dependencyCount` is the number of dependencies between pairs of subpasses.
- `pDependencies` is a pointer to an array of `dependencyCount` `VkSubpassDependency2` structures describing dependencies between pairs of subpasses.
- `correlatedViewMaskCount` is the number of correlation masks.
- `pCorrelatedViewMasks` is a pointer to an array of view masks indicating sets of views that may be more efficient to render concurrently.

Parameters defined by this structure with the same name as those in `VkRenderPassCreateInfo` have the identical effect to those parameters; the child structures are variants of those used in `VkRenderPassCreateInfo` which add `sType` and `pNext` parameters, allowing them to be extended.

If the `VkSubpassDescription2::viewMask` member of any element of `pSubpasses` is not zero, `multiview` functionality is considered to be enabled for this render pass.
correlatedViewMaskCount and pCorrelatedViewMasks have the same effect as VkRenderPassMultiviewCreateInfo::correlationMaskCount and VkRenderPassMultiviewCreateInfo::pCorrelationMasks, respectively.
Valid Usage

- **VUID-VkRenderPassCreateInfo2-None-03049**
  If any two subpasses operate on attachments with overlapping ranges of the same `VkDeviceMemory` object, and at least one subpass writes to that area of `VkDeviceMemory`, a subpass dependency **must** be included (either directly or via some intermediate subpasses) between them.

- **VUID-VkRenderPassCreateInfo2-attachment-03050**
  If the attachment member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments`, or `pDepthStencilAttachment`, or the attachment indexed by any element of `pPreserveAttachments` in any given element of `pSubpasses` is bound to a range of a `VkDeviceMemory` object that overlaps with any other attachment in any subpass (including the same subpass), the `VkAttachmentDescription2` structures describing them **must** include `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` in flags.

- **VUID-VkRenderPassCreateInfo2-attachment-03051**
  If the attachment member of any element of `pInputAttachments`, `pColorAttachments`, `pResolveAttachments`, or `pDepthStencilAttachment`, or any element of `pPreserveAttachments` in any given element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, it **must** be less than `attachmentCount`.

- **VUID-VkRenderPassCreateInfo2-pAttachments-02522**
  For any member of `pAttachments` with a `loadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`.

- **VUID-VkRenderPassCreateInfo2-pAttachments-02523**
  For any member of `pAttachments` with a `stencilLoadOp` equal to `VK_ATTACHMENT_LOAD_OP_CLEAR`, the first use of that attachment **must** not specify a `layout` equal to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkRenderPassCreateInfo2-pDependencies-03054**
  For any element of `pDependencies`, if the `srcSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `srcStageMask` member of that dependency **must** be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the source subpass.

- **VUID-VkRenderPassCreateInfo2-pDependencies-03055**
  For any element of `pDependencies`, if the `dstSubpass` is not `VK_SUBPASS_EXTERNAL`, all stage flags included in the `dstStageMask` member of that dependency **must** be a pipeline stage supported by the pipeline identified by the `pipelineBindPoint` member of the destination subpass.

- **VUID-VkRenderPassCreateInfo2-pCorrelatedViewMasks-03056**
  The set of bits included in any element of `pCorrelatedViewMasks` **must** not overlap with the set of bits included in any other element of `pCorrelatedViewMasks`.

- **VUID-VkRenderPassCreateInfo2-viewMask-03057**

Chapter 8. Render Pass | 421
If the `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` is 0, `correlatedViewMaskCount` must be 0

- **VUID-VkRenderPassCreateInfo2-viewMask-03058**
  The `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` must either all be 0, or all not be 0

- **VUID-VkRenderPassCreateInfo2-viewMask-03059**
  If the `VkSubpassDescription2::viewMask` member of all elements of `pSubpasses` is 0, the `dependencyFlags` member of any element of `pDependencies` must not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`

- **VUID-VkRenderPassCreateInfo2-pDependencies-03060**
  For any element of `pDependencies` where its `srcSubpass` member equals its `dstSubpass` member, if the `viewMask` member of the corresponding element of `pSubpasses` includes more than one bit, its `dependencyFlags` member must include `VK_DEPENDENCY_VIEW_LOCAL_BIT`

- **VUID-VkRenderPassCreateInfo2-attachment-02525**
  If the `attachment` member of any element of the `pInputAttachments` member of any element of `pSubpasses` is not `VK_ATTACHMENT_UNUSED`, the `aspectMask` member of that element of `pInputAttachments` must only include aspects that are present in images of the format specified by the element of `pAttachments` specified by `attachment`

- **VUID-VkRenderPassCreateInfo2-srcSubpass-02526**
  The `srcSubpass` member of each element of `pDependencies` must be less than `subpassCount`

- **VUID-VkRenderPassCreateInfo2-dstSubpass-02527**
  The `dstSubpass` member of each element of `pDependencies` must be less than `subpassCount`

- **VUID-VkRenderPassCreateInfo2-pAttachments-04585**
  If any element of `pAttachments` is used as a fragment shading rate attachment in any subpass, it must not be used as any other attachment in the render pass

- **VUID-VkRenderPassCreateInfo2-flags-04521**
  If `flags` includes `VK_RENDER_PASS_CREATE_TRANSFORM_BIT_QCOM`, an element of `pSubpasses` includes an instance of `VkFragmentShadingRateAttachmentInfoKHR` in its `pNext` chain, and the `pFragmentShadingRateAttachment` member of that structure is not equal to NULL, the `attachment` member of `pFragmentShadingRateAttachment` must be `VK_ATTACHMENT_UNUSED`

- **VUID-VkRenderPassCreateInfo2-pAttachments-04586**
  If any element of `pAttachments` is used as a fragment shading rate attachment in any subpass, it must have an image format whose potential format features contain `VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`
Valid Usage (Implicit)

- **VUID-VkRenderPassCreateInfo2-sType-sType**
  
  *sType must be* \texttt{VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO_2}.

- **VUID-VkRenderPassCreateInfo2-pNext-pNext**
  
  *pNext must be* \texttt{NULL} or a pointer to a valid instance of \texttt{VkRenderPassFragmentDensityMapCreateInfoEXT}.

- **VUID-VkRenderPassCreateInfo2-sType-unique**
  
  The *sType value of each struct in the pNext chain must be unique*.

- **VUID-VkRenderPassCreateInfo2-flags-parameter**
  
  *flags must be a valid combination of* \texttt{VkRenderPassCreateFlagBits} values.

- **VUID-VkRenderPassCreateInfo2-pAttachments-parameter**
  
  If *attachmentCount is not 0, pAttachments must be a valid pointer to an array of attachmentCount valid* \texttt{VkAttachmentDescription2} structures.

- **VUID-VkRenderPassCreateInfo2-pSubpasses-parameter**
  
  *pSubpasses must be a valid pointer to an array of subpassCount valid* \texttt{VkSubpassDescription2} structures.

- **VUID-VkRenderPassCreateInfo2-pDependencies-parameter**
  
  If *dependencyCount is not 0, pDependencies must be a valid pointer to an array of dependencyCount valid* \texttt{VkSubpassDependency2} structures.

- **VUID-VkRenderPassCreateInfo2-pCorrelatedViewMasks-parameter**
  
  If *correlatedViewMaskCount is not 0, pCorrelatedViewMasks must be a valid pointer to an array of correlatedViewMaskCount uint32_t values*.

- **VUID-VkRenderPassCreateInfo2-subpassCount-arraylength**
  
  *subpassCount must be greater than 0*.

The \texttt{VkAttachmentDescription2} structure is defined as:

```c
typedef struct VkAttachmentDescription2 {
    VkStructureType sType;
    const void* pNext;
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
    VkAttachmentLoadOp stencilLoadOp;
    VkAttachmentStoreOp stencilStoreOp;
    VkImageLayout initialLayout;
    VkImageLayout finalLayout;
} VkAttachmentDescription2;
```

or the equivalent
// Provided by VK_KHR_create_renderpass2

typedef VkAttachmentDescription2 VkAttachmentDescription2KHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkAttachmentDescriptionFlagBits** specifying additional properties of the attachment.
- **format** is a **VkFormat** value specifying the format of the image that will be used for the attachment.
- **samples** is the number of samples of the image as defined in **VkSampleCountFlagBits**.
- **loadOp** is a **VkAttachmentLoadOp** value specifying how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used.
- **storeOp** is a **VkAttachmentStoreOp** value specifying how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used.
- **stencilLoadOp** is a **VkAttachmentLoadOp** value specifying how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used.
- **stencilStoreOp** is a **VkAttachmentStoreOp** value specifying how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used.
- **initialLayout** is the layout the attachment image subresource will be in when a render pass instance begins.
- **finalLayout** is the layout the attachment image subresource will be transitioned to when a render pass instance ends.

Parameters defined by this structure with the same name as those in **VkAttachmentDescription** have the identical effect to those parameters.

If the **separateDepthStencilLayouts** feature is enabled, and **format** is a depth/stencil format, **initialLayout** and **finalLayout** can be set to a layout that only specifies the layout of the depth aspect.

If the **pNext** chain includes a **VkAttachmentDescriptionStencilLayout** structure, then the **stencilInitialLayout** and **stencilFinalLayout** members specify the initial and final layouts of the stencil aspect of a depth/stencil format, and **initialLayout** and **finalLayout** only apply to the depth aspect. For depth-only formats, the **VkAttachmentDescriptionStencilLayout** structure is ignored. For stencil-only formats, the initial and final layouts of the stencil aspect are taken from the **VkAttachmentDescriptionStencilLayout** structure if present, or **initialLayout** and **finalLayout** if not present.

If **format** is a depth/stencil format, and either **initialLayout** or **finalLayout** does not specify a layout for the stencil aspect, then the application **must** specify the initial and final layouts of the stencil aspect by including a **VkAttachmentDescriptionStencilLayout** structure in the **pNext** chain.
Valid Usage

• VUID-VkAttachmentDescription2-finalLayout-03061
  finalLayout must not be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED

• VUID-VkAttachmentDescription2-format-03294
  If format is a color format, initialLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

• VUID-VkAttachmentDescription2-format-03295
  If format is a depth/stencil format, initialLayout must not be
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

• VUID-VkAttachmentDescription2-format-03296
  If format is a color format, finalLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

• VUID-VkAttachmentDescription2-format-03297
  If format is a depth/stencil format, finalLayout must not be
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

• VUID-VkAttachmentDescription2-separateDepthStencilLayouts-03298
  If the separateDepthStencilLayouts feature is not enabled, initialLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL
  or
  VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkAttachmentDescription2-separateDepthStencilLayouts-03299
  If the separateDepthStencilLayouts feature is not enabled, finalLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL
  or
  VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkAttachmentDescription2-format-03300
  If format is a color format, initialLayout must not be
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL
  or
  VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL

• VUID-VkAttachmentDescription2-format-03301
  If format is a depth/stencil format which includes both depth and stencil aspects, and
  initialLayout is
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL
  or
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, the pNext chain must include a
**VkAttachmentDescriptionStencilLayout** structure

- **VUID-VkAttachmentDescription2-format-03303**
  If `format` is a depth/stencil format which includes both depth and stencil aspects, and `finalLayout` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, the `pNext` chain **must** include a `VkAttachmentDescriptionStencilLayout` structure.

- **VUID-VkAttachmentDescription2-format-03304**
  If `format` is a depth/stencil format which includes only the depth aspect, `initialLayout` **must not** be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-03305**
  If `format` is a depth/stencil format which includes only the depth aspect, `finalLayout` **must not** be `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-03306**
  If `format` is a depth/stencil format which includes only the stencil aspect, `initialLayout` **must not** be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.

- **VUID-VkAttachmentDescription2-format-03307**
  If `format` is a depth/stencil format which includes only the stencil aspect, `finalLayout` **must not** be `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`.
Valid Usage (Implicit)

- VUID-VkAttachmentDescription2-sType-sType
  
  **sType must** be **VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_2**

- VUID-VkAttachmentDescription2-pNext-pNext
  
  **pNext must** be **NULL** or a pointer to a valid instance of
  
  VkAttachmentDescriptionStencilLayout

- VUID-VkAttachmentDescription2-sType-unique
  
  The **sType** value of each struct in the **pNext** chain must be **unique**

- VUID-VkAttachmentDescription2-flags-parameter
  
  **flags must** be a valid combination of **VkAttachmentDescriptionFlagBits** values

- VUID-VkAttachmentDescription2-format-parameter
  
  **format must** be a valid **VkFormat** value

- VUID-VkAttachmentDescription2-samples-parameter
  
  **samples must** be a valid **VkSampleCountFlagBits** value

- VUID-VkAttachmentDescription2-loadOp-parameter
  
  **loadOp must** be a valid **VkAttachmentLoadOp** value

- VUID-VkAttachmentDescription2-storeOp-parameter
  
  **storeOp must** be a valid **VkAttachmentStoreOp** value

- VUID-VkAttachmentDescription2-stencilLoadOp-parameter
  
  **stencilLoadOp must** be a valid **VkAttachmentLoadOp** value

- VUID-VkAttachmentDescription2-stencilStoreOp-parameter
  
  **stencilStoreOp must** be a valid **VkAttachmentStoreOp** value

- VUID-VkAttachmentDescription2-initialLayout-parameter
  
  **initialLayout must** be a valid **VkImageLayout** value

- VUID-VkAttachmentDescription2-finalLayout-parameter
  
  **finalLayout must** be a valid **VkImageLayout** value

The **VkAttachmentDescriptionStencilLayout** structure is defined as:

```c
typedef struct VkAttachmentDescriptionStencilLayout {
    VkStructureType    sType;
    void*              pNext;
    VkImageLayout      stencilInitialLayout;
    VkImageLayout      stencilFinalLayout;
} VkAttachmentDescriptionStencilLayout;
```

or the equivalent

```c
// Provided by VK_KHR_separate_depth_stencil_layouts
typedef VkAttachmentDescriptionStencilLayout VkAttachmentDescriptionStencilLayoutKHR;
```
• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **stencilInitialLayout** is the layout the stencil aspect of the attachment image subresource will be in when a render pass instance begins.

• **stencilFinalLayout** is the layout the stencil aspect of the attachment image subresource will be transitioned to when a render pass instance ends.

## Valid Usage

- **VUID-VkAttachmentDescriptionStencilLayout-stencilInitialLayout-03308**
  ```
  stencilInitialLayout must not be
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, or
  ```

- **VUID-VkAttachmentDescriptionStencilLayout-stencilFinalLayout-03309**
  ```
  stencilFinalLayout must not be
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL,
  VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, or
  ```

- **VUID-VkAttachmentDescriptionStencilLayout-stencilFinalLayout-03310**
  ```
  stencilFinalLayout must not be
  VK_IMAGE_LAYOUT_UNDEFINED or
  VK_IMAGE_LAYOUT_PREINITIALIZED
  ```

## Valid Usage (Implicit)

- **VUID-VkAttachmentDescriptionStencilLayout-sType-sType**
  ```
  sType must be
  VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT
  ```

- **VUID-VkAttachmentDescriptionStencilLayout-stencilInitialLayout-parameter**
  ```
  stencilInitialLayout must be a valid VkImageLayout value
  ```

- **VUID-VkAttachmentDescriptionStencilLayout-stencilFinalLayout-parameter**
  ```
  stencilFinalLayout must be a valid VkImageLayout value
  ```

The **VkSubpassDescription2** structure is defined as:
typedef struct VkSubpassDescription2 {
    VkStructureType                  sType;
    const void*                      pNext;
    VkSubpassDescriptionFlags        flags;
    VkPipelineBindPoint              pipelineBindPoint;
    uint32_t                         viewMask;
    uint32_t                         inputAttachmentCount;
    const VkAttachmentReference2*    pInputAttachments;
    uint32_t                         colorAttachmentCount;
    const VkAttachmentReference2*    pColorAttachments;
    const VkAttachmentReference2*    pResolveAttachments;
    const VkAttachmentReference2*    pDepthStencilAttachment;
    uint32_t                         preserveAttachmentCount;
    const uint32_t*                  pPreserveAttachments;
} VkSubpassDescription2;

or the equivalent

// Provided by VK_KHR_create_renderpass2
typedef VkSubpassDescription2 VkSubpassDescription2KHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkSubpassDescriptionFlagBits specifying usage of the subpass.
- **pipelineBindPoint** is a VkPipelineBindPoint value specifying the pipeline type supported for this subpass.
- **viewMask** is a bitfield of view indices describing which views rendering is broadcast to in this subpass, when multiview is enabled.
- **inputAttachmentCount** is the number of input attachments.
- **pInputAttachments** is a pointer to an array of VkAttachmentReference2 structures defining the input attachments for this subpass and their layouts.
- **colorAttachmentCount** is the number of color attachments.
- **pColorAttachments** is a pointer to an array of VkAttachmentReference2 structures defining the color attachments for this subpass and their layouts.
- **pResolveAttachments** is an optional array of colorAttachmentCount VkAttachmentReference2 structures defining the resolve attachments for this subpass and their layouts.
- **pDepthStencilAttachment** is a pointer to a VkAttachmentReference2 structure specifying the depth/stencil attachment for this subpass and its layout.
- **preserveAttachmentCount** is the number of preserved attachments.
- **pPreserveAttachments** is a pointer to an array of preserveAttachmentCount render pass attachment indices identifying attachments that are not used by this subpass, but whose contents must be preserved throughout the subpass.
Parameters defined by this structure with the same name as those in `VkSubpassDescription` have the identical effect to those parameters.

`viewMask` has the same effect for the described subpass as `VkRenderPassMultiviewCreateInfo::pViewMasks` has on each corresponding subpass.

If an instance of `VkFragmentShadingRateAttachmentInfoKHR` is included in the `pNext` chain, `pFragmentShadingRateAttachment` is not `NULL`, and its `attachment` member is not `VK_ATTACHMENT_UNUSED`, the identified attachment defines a fragment shading rate attachment for that subpass.
Valid Usage

• VUID-VkSubpassDescription2-pipelineBindPoint-03062
  
  pipelineBindPoint must be VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-VkSubpassDescription2-colorAttachmentCount-03063
  
  colorAttachmentCount must be less than or equal to VkPhysicalDeviceLimits::maxColorAttachments

• VUID-VkSubpassDescription2-loadOp-03064
  
  If the first use of an attachment in this render pass is as an input attachment, and the attachment is not also used as a color or depth/stencil attachment in the same subpass, then loadOp must not be VK_ATTACHMENT_LOAD_OP_CLEAR

• VUID-VkSubpassDescription2-pResolveAttachments-03065
  
  If pResolveAttachments is not NULL, for each resolve attachment that does not have the value VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have the value VK_ATTACHMENT_UNUSED

• VUID-VkSubpassDescription2-pResolveAttachments-03066
  
  If pResolveAttachments is not NULL, for each resolve attachment that is not VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescription2-pResolveAttachments-03067
  
  If pResolveAttachments is not NULL, each resolve attachment that is not VK_ATTACHMENT_UNUSED must have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescription2-pResolveAttachments-03068
  
  Any given element of pResolveAttachments must have the same VkFormat as its corresponding color attachment

• VUID-VkSubpassDescription2-pColorAttachments-03069
  
  All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have the same sample count

• VUID-VkSubpassDescription2-pInputAttachments-02897
  
  All attachments in pInputAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain at least VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT or VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkSubpassDescription2-pColorAttachments-02898
  
  All attachments in pColorAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

• VUID-VkSubpassDescription2-pResolveAttachments-02899
  
  All attachments in pResolveAttachments that are not VK_ATTACHMENT_UNUSED must have image formats whose potential format features contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

• VUID-VkSubpassDescription2-pDepthStencilAttachment-02900
  
  If pDepthStencilAttachment is not NULL and the attachment is not VK_ATTACHMENT_UNUSED then it must have an image format whose potential format features contain VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
If the `VK_AMD_mixed_attachment_samples` extension is enabled, all attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` must have a sample count that is smaller than or equal to the sample count of `pDepthStencilAttachment` if it is not `VK_ATTACHMENT_UNUSED`.

If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `pDepthStencilAttachment` is not `VK_ATTACHMENT_UNUSED` and any attachments in `pColorAttachments` are not `VK_ATTACHMENT_UNUSED`, they must have the same sample count.

The attachment member of any element of `pPreserveAttachments` must not be `VK_ATTACHMENT_UNUSED`.

Any given element of `pPreserveAttachments` must not also be an element of any other member of the subpass description.

If any attachment is used by more than one `VkAttachmentReference` member, then each use must use the same layout.

Attachments must follow the image layout requirements based on the type of attachment it is being used as.

If `flags` includes `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX`, it must also include `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX`.

If the attachment member of any element of `pInputAttachments` is not `VK_ATTACHMENT_UNUSED`, then the `aspectMask` member must be a valid combination of `VkImageAspectFlagBits`.

If the attachment member of any element of `pInputAttachments` is not `VK_ATTACHMENT_UNUSED`, then the `aspectMask` member must not be 0.

If the attachment member of any element of `pInputAttachments` is not `VK_ATTACHMENT_UNUSED`, then the `aspectMask` member must not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`.

An attachment must not be used in both `pDepthStencilAttachment` and `pColorAttachments`.
Valid Usage (Implicit)

- VUID-VkSubpassDescription2-sType-sType
  sType must be VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_2

- VUID-VkSubpassDescription2-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkFragmentShadingRateAttachmentInfoKHR or VkSubpassDescriptionDepthStencilResolve

- VUID-VkSubpassDescription2-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkSubpassDescription2-flags-parameter
  flags must be a valid combination of VkSubpassDescriptionFlagBits values

- VUID-VkSubpassDescription2-pipelineBindPoint-parameter
  pipelineBindPoint must be a valid VkPipelineBindPoint value

- VUID-VkSubpassDescription2-pInputAttachments-parameter
  If inputAttachmentCount is not 0, pInputAttachments must be a valid pointer to an array of inputAttachmentCount valid VkAttachmentReference2 structures

- VUID-VkSubpassDescription2-pColorAttachments-parameter
  If colorAttachmentCount is not 0, pColorAttachments must be a valid pointer to an array of colorAttachmentCount valid VkAttachmentReference2 structures

- VUID-VkSubpassDescription2-pResolveAttachments-parameter
  If colorAttachmentCount is not 0, and pResolveAttachments is not NULL, pResolveAttachments must be a valid pointer to an array of colorAttachmentCount valid VkAttachmentReference2 structures

- VUID-VkSubpassDescription2-pDepthStencilAttachment-parameter
  If pDepthStencilAttachment is not NULL, pDepthStencilAttachment must be a valid pointer to a valid VkAttachmentReference2 structure

- VUID-VkSubpassDescription2-pPreserveAttachments-parameter
  If preserveAttachmentCount is not 0, pPreserveAttachments must be a valid pointer to an array of preserveAttachmentCount uint32_t values

If the pNext list of VkSubpassDescription2 includes a VkSubpassDescriptionDepthStencilResolve structure, then that structure describes multisample resolve operations for the depth/stencil attachment in a subpass.

The VkSubpassDescriptionDepthStencilResolve structure is defined as:
typedef struct VkSubpassDescriptionDepthStencilResolve {
    VkStructureType                  sType;
    const void*                      pNext;
    VkResolveModeFlagBits            depthResolveMode;
    VkResolveModeFlagBits            stencilResolveMode;
    const VkAttachmentReference2*    pDepthStencilResolveAttachment;
} VkSubpassDescriptionDepthStencilResolve;

or the equivalent

// Provided by VK_KHR_depth_stencil_resolve
typedef VkSubpassDescriptionDepthStencilResolve
VkSubpassDescriptionDepthStencilResolveKHR;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `depthResolveMode` is a bitmask of `VkResolveModeFlagBits` describing the depth resolve mode.
- `stencilResolveMode` is a bitmask of `VkResolveModeFlagBits` describing the stencil resolve mode.
- `pDepthStencilResolveAttachment` is an optional `VkAttachmentReference` structure defining the depth/stencil resolve attachment for this subpass and its layout.
Valid Usage

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03177
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, pDepthStencilAttachment must not have the value VK_ATTACHMENT_UNUSED

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03178
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, depthResolveMode and stencilResolveMode must not both be VK_RESOLVE_MODE_NONE

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03179
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, pDepthStencilAttachment must not have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03180
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED, pDepthStencilResolveAttachment must have a sample count of VK_SAMPLE_COUNT_1_BIT

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-02651
  If pDepthStencilResolveAttachment is not NULL and does not have the value VK_ATTACHMENT_UNUSED then it must have a format whose features contain VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03181
  If the VkFormat of pDepthStencilResolveAttachment has a depth component, then the VkFormat of pDepthStencilAttachment must have a depth component with the same number of bits and numerical type

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03182
  If the VkFormat of pDepthStencilResolveAttachment has a stencil component, then the VkFormat of pDepthStencilAttachment must have a stencil component with the same number of bits and numerical type

• VUID-VkSubpassDescriptionDepthStencilResolve-depthResolveMode-03183
  The value of depthResolveMode must be one of the bits set in VkPhysicalDeviceDepthStencilResolveProperties::supportedDepthResolveModes or VK_RESOLVE_MODE_NONE

• VUID-VkSubpassDescriptionDepthStencilResolve-stencilResolveMode-03184
  The value of stencilResolveMode must be one of the bits set in VkPhysicalDeviceDepthStencilResolveProperties::supportedStencilResolveModes or VK_RESOLVE_MODE_NONE

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03185
  If the VkFormat of pDepthStencilResolveAttachment has both depth and stencil components, VkPhysicalDeviceDepthStencilResolveProperties::independentResolve is VK_FALSE, and VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone is VK_FALSE, then the values of depthResolveMode and stencilResolveMode must be identical

• VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-03186
  If the VkFormat of pDepthStencilResolveAttachment has both depth and stencil components, VkPhysicalDeviceDepthStencilResolveProperties::independentResolve is VK_FALSE, and VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone is VK_FALSE, then the values of depthResolveMode and stencilResolveMode must be identical
components, `VkPhysicalDeviceDepthStencilResolveProperties::independentResolve` is `VK_FALSE` and `VkPhysicalDeviceDepthStencilResolveProperties::independentResolveNone` is `VK_TRUE`, then the values of `depthResolveMode` and `stencilResolveMode` must be identical or one of them must be `VK_RESOLVE_MODE_NONE`.

- VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-04588
  If the `VkFormat` of `pDepthStencilResolveAttachment` has a depth component, `depthResolveMode` must be a valid `VkResolveModeFlagBits` value.

- VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-04589
  If the `VkFormat` of `pDepthStencilResolveAttachment` has a stencil component, `stencilResolveMode` must be a valid `VkResolveModeFlagBits` value.

### Valid Usage (Implicit)

- VUID-VkSubpassDescriptionDepthStencilResolve-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE`.

- VUID-VkSubpassDescriptionDepthStencilResolve-pDepthStencilResolveAttachment-parameter
  If `pDepthStencilResolveAttachment` is not NULL, `pDepthStencilResolveAttachment` must be a valid pointer to a valid `VkAttachmentReference2` structure.

Possible values of `VkSubpassDescriptionDepthStencilResolve::depthResolveMode` and `stencilResolveMode`, specifying the depth and stencil resolve modes, are:

```c
typedef enum VkResolveModeFlagBits {  
  VK_RESOLVE_MODE_NONE = 0,  
  VK_RESOLVE_MODE_SAMPLE_ZERO_BIT = 0x00000001,  
  VK_RESOLVE_MODE_AVERAGE_BIT = 0x00000002,  
  VK_RESOLVE_MODE_MIN_BIT = 0x00000004,  
  VK_RESOLVE_MODE_MAX_BIT = 0x00000008,  
  // Provided by VK_KHR_depth_stencilResolve
  VK_RESOLVE_MODE_NONE_KHR = VK_RESOLVE_MODE_NONE,  
  // Provided by VK_KHR_depth_stencilResolve
  VK_RESOLVE_MODE_SAMPLE_ZERO_BIT_KHR = VK_RESOLVE_MODE_SAMPLE_ZERO_BIT,  
  // Provided by VK_KHR_depth_stencilResolve
  VK_RESOLVE_MODE_AVERAGE_BIT_KHR = VK_RESOLVE_MODE_AVERAGE_BIT,  
  // Provided by VK_KHR_depth_stencil_resolve
  VK_RESOLVE_MODE_MIN_BIT_KHR = VK_RESOLVE_MODE_MIN_BIT,  
  // Provided by VK_KHR_depth_stencil_resolve
  VK_RESOLVE_MODE_MAX_BIT_KHR = VK_RESOLVE_MODE_MAX_BIT,  
} VkResolveModeFlagBits;
```

or the equivalent:

```c
// Provided by VK_KHR_depth_stencil_resolve
typedef VkResolveModeFlagBits VkResolveModeFlagBitsKHR;
```
• **VK_RESOLVE_MODE_NONE** indicates that no resolve operation is done.
• **VK_RESOLVE_MODE_SAMPLE_ZERO_BIT** indicates that result of the resolve operation is equal to the value of sample 0.
• **VK_RESOLVE_MODE_AVERAGE_BIT** indicates that result of the resolve operation is the average of the sample values.
• **VK_RESOLVE_MODE_MIN_BIT** indicates that result of the resolve operation is the minimum of the sample values.
• **VK_RESOLVE_MODE_MAX_BIT** indicates that result of the resolve operation is the maximum of the sample values.

```c
typedef VkFlags VkResolveModeFlags;
```
or the equivalent

```c
// Provided by VK_KHR_depth_stencil_resolve
typedef VkResolveModeFlags VkResolveModeFlagsKHR;
```

**VkResolveModeFlags** is a bitmask type for setting a mask of zero or more **VkResolveModeFlagBits**.

The **VkFragmentShadingRateAttachmentInfoKHR** structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkFragmentShadingRateAttachmentInfoKHR {
    VkStructureType           sType;
    const void*               pNext;
    const VkAttachmentReference2*  pFragmentShadingRateAttachment;
    VkExtent2D                shadingRateAttachmentTexelSize;
} VkFragmentShadingRateAttachmentInfoKHR;
```

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **pFragmentShadingRateAttachment** is an optional **VkAttachmentReference2KHR** structure defining the fragment shading rate attachment for this subpass.
• **shadingRateAttachmentTexelSize** specifies the size of the portion of the framebuffer corresponding to each texel in **pFragmentShadingRateAttachment**.

If no shading rate attachment is specified, or if this structure isn’t specified, the implementation behaves as if a valid shading rate attachment was specified with all texels specifying a single pixel per fragment.
Valid Usage

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04524**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, its layout member must be equal to `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR`.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04525**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, `shadingRateAttachmentTexelSize.width` must be a power of two.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04526**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, `shadingRateAttachmentTexelSize.width` must be less than or equal to `maxFragmentShadingRateAttachmentTexelSize.width`.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04527**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, `shadingRateAttachmentTexelSize.width` must be greater than or equal to `minFragmentShadingRateAttachmentTexelSize.width`.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04528**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, `shadingRateAttachmentTexelSize.height` must be a power of two.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04529**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, `shadingRateAttachmentTexelSize.height` must be less than or equal to `maxFragmentShadingRateAttachmentTexelSize.height`.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04530**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, `shadingRateAttachmentTexelSize.height` must be greater than or equal to `minFragmentShadingRateAttachmentTexelSize.height`.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04531**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, the quotient of `shadingRateAttachmentTexelSize.width` and `shadingRateAttachmentTexelSize.height` must be less than or equal to `maxFragmentShadingRateAttachmentTexelSizeAspectRatio`.

- **VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-04532**: If `pFragmentShadingRateAttachment` is not `NULL` and its attachment member is not `VK_ATTACHMENT_UNUSED`, the quotient of `shadingRateAttachmentTexelSize.height` and `shadingRateAttachmentTexelSize.width` must be less than or equal to `maxFragmentShadingRateAttachmentTexelSizeAspectRatio`.

---

438 | Chapter 8. Render Pass
Valid Usage (Implicit)

- VUID-VkFragmentShadingRateAttachmentInfoKHR-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_FRAGMENT_SHADING_RATE_ATTACHMENT_INFO_KHR`

- VUID-VkFragmentShadingRateAttachmentInfoKHR-pFragmentShadingRateAttachment-parameter

  **pFragmentShadingRateAttachment** must be a valid pointer to a valid
  
  `VkAttachmentReference2` structure

The `VkAttachmentReference2` structure is defined as:

```c
typedef struct VkAttachmentReference2 {
    VkStructureType       sType;
    const void*           pNext;
    uint32_t              attachment;
    VkImageLayout         layout;
    VkImageAspectFlags    aspectMask;
} VkAttachmentReference2;
```

or the equivalent

```c
// Provided by VK_KHR_create_renderpass2
typedef VkAttachmentReference2 VkAttachmentReference2KHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **attachment** is either an integer value identifying an attachment at the corresponding index in
  
  `VkRenderPassCreateInfo::pAttachments`, or `VK_ATTACHMENT_UNUSED` to signify that this attachment is
  
  not used.
- **layout** is a `VkImageLayout` value specifying the layout the attachment uses during the subpass.
- **aspectMask** is a mask of which aspect(s) can be accessed within the specified subpass as an input
  
  attachment.

Parameters defined by this structure with the same name as those in `VkAttachmentReference` have
the identical effect to those parameters.

**aspectMask** is ignored when this structure is used to describe anything other than an input
attachment reference.

If the `separateDepthStencilLayouts` feature is enabled, and **attachment** has a depth/stencil format,

**layout** can be set to a layout that only specifies the layout of the depth aspect.

If **layout** only specifies the layout of the depth aspect of the attachment, the layout of the stencil
aspect is specified by the **stencillayout** member of a `VkAttachmentReferenceStencilLayout`
structure included in the **pNext** chain. Otherwise, **layout** describes the layout for all relevant image
Valid Usage

- **VUID-VkAttachmentReference2-layout-03077**
  If attachment is not VK_ATTACHMENT_UNUSED, layout must not be VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_PREINITIALIZED, or VK_IMAGE_LAYOUT_PRESENT_SRC_KHR.

- **VUID-VkAttachmentReference2-separateDepthStencilLayouts-03313**
  If the separateDepthStencils Layout feature is not enabled, and attachment is not VK_ATTACHMENT_UNUSED, layout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkAttachmentReference2-attachment-04754**
  If attachment is not VK_ATTACHMENT_UNUSED, and the format of the referenced attachment is a color format, layout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkAttachmentReference2-attachment-04755**
  If attachment is not VK_ATTACHMENT_UNUSED, and the format of the referenced attachment is a depth/stencil format which includes both depth and stencil aspects, and layout is VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, the pNext chain must include a VkAttachmentReferenceStencilLayout structure.

- **VUID-VkAttachmentReference2-attachment-04756**
  If attachment is not VK_ATTACHMENT_UNUSED, and the format of the referenced attachment is a depth/stencil format which includes only the depth aspect, layout must not be VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

- **VUID-VkAttachmentReference2-attachment-04757**
  If attachment is not VK_ATTACHMENT_UNUSED, and the format of the referenced attachment is a depth/stencil format which includes only the stencil aspect, layout must not be VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL.

Valid Usage (Implicit)

- **VUID-VkAttachmentReference2-sType-sType**
  sType must be VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_2.

- **VUID-VkAttachmentReference2-pNext-pNext**
  pNext must be NULL or a pointer to a valid instance of VkAttachmentReferenceStencilLayout.

- **VUID-VkAttachmentReference2-sType-unique**
  The sType value of each struct in the pNext chain must be unique.

- **VUID-VkAttachmentReference2-layout-parameter**
  layout must be a valid VkImageLayout value.
The `VkAttachmentReferenceStencilLayout` structure is defined as:

```c
typedef struct VkAttachmentReferenceStencilLayout {
    VkStructureType sType;
    void* pNext;
    VkImageLayout stencilLayout;
} VkAttachmentReferenceStencilLayout;
```

or the equivalent

```c
// Provided by VK_KHR_separate_depth_stencil_layouts
typedef VkAttachmentReferenceStencilLayout VkAttachmentReferenceStencilLayoutKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `stencilLayout` is a `VkImageLayout` value specifying the layout the stencil aspect of the attachment uses during the subpass.

### Valid Usage

- VUID-VkAttachmentReferenceStencilLayout-stencilLayout-03318
  
  `stencilLayout` must not be `VK_IMAGE_LAYOUT_UNDEFINED`, `VK_IMAGE_LAYOUT_PREINITIALIZED`, `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` or

### Valid Usage (Implicit)

- VUID-VkAttachmentReferenceStencilLayout-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT`

- VUID-VkAttachmentReferenceStencilLayout-stencilLayout-parameter
  
  `stencilLayout` must be a valid `VkImageLayout` value

The `VkSubpassDependency2` structure is defined as:

Chapter 8. Render Pass | 441
typedef struct VkSubpassDependency2 {
    VkStructureType         sType;
    const void*             pNext;
    uint32_t                srcSubpass;
    uint32_t                dstSubpass;
    VkPipelineStageFlags    srcStageMask;
    VkPipelineStageFlags    dstStageMask;
    VkAccessFlags           srcAccessMask;
    VkAccessFlags           dstAccessMask;
    VkDependencyFlags       dependencyFlags;
    int32_t                 viewOffset;
} VkSubpassDependency2;

or the equivalent

// Provided by VK_KHR_create_renderpass2
typedef VkSubpassDependency2 VkSubpassDependency2KHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcSubpass** is the subpass index of the first subpass in the dependency, or **VK_SUBPASS_EXTERNAL**.
- **dstSubpass** is the subpass index of the second subpass in the dependency, or **VK_SUBPASS_EXTERNAL**.
- **srcStageMask** is a bitmask of **VkPipelineStageFlagBits** specifying the source stage mask.
- **dstStageMask** is a bitmask of **VkPipelineStageFlagBits** specifying the destination stage mask.
- **srcAccessMask** is a bitmask of **VkAccessFlagBits** specifying a source access mask.
- **dstAccessMask** is a bitmask of **VkAccessFlagBits** specifying a destination access mask.
- **dependencyFlags** is a bitmask of **VkDependencyFlagBits**.
- **viewOffset** controls which views in the source subpass the views in the destination subpass depend on.

Parameters defined by this structure with the same name as those in **VkSubpassDependency** have the identical effect to those parameters.

**viewOffset** has the same effect for the described subpass dependency as **VkRenderPassMultiviewCreateInfo::pViewOffsets** has on each corresponding subpass dependency.

If an instance of **VkMemoryBarrier2KHR** is included in the **pNext** chain, **srcStageMask**, **dstStageMask**, **srcAccessMask**, and **dstAccessMask** parameters are ignored. The synchronization and access scopes instead are defined by the parameters of **VkMemoryBarrier2KHR**.
Valid Usage

- VUID-VkSubpassDependency2-srcStageMask-03080
  If the geometry shaders feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-VkSubpassDependency2-dstStageMask-03081
  If the geometry shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-VkSubpassDependency2-srcStageMask-03082
  If the tessellation shaders feature is not enabled, srcStageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-VkSubpassDependency2-dstStageMask-03083
  If the tessellation shaders feature is not enabled, dstStageMask must not contain VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-VkSubpassDependency2-srcSubpass-03084
  srcSubpass must be less than or equal to dstSubpass, unless one of them is VK_SUBPASS_EXTERNAL, to avoid cyclic dependencies and ensure a valid execution order

- VUID-VkSubpassDependency2-srcSubpass-03085
  srcSubpass and dstSubpass must not both be equal to VK_SUBPASS_EXTERNAL

- VUID-VkSubpassDependency2-srcSubpass-03087
  If srcSubpass is equal to dstSubpass and not all of the stages in srcStageMask and dstStageMask are framebuffer-space stages, the logically latest pipeline stage in srcStageMask must be logically earlier than or equal to the logically earliest pipeline stage in dstStageMask

- VUID-VkSubpassDependency2-srcAccessMask-03088
  Any access flag included in srcAccessMask must be supported by one of the pipeline stages in srcStageMask, as specified in the table of supported access types

- VUID-VkSubpassDependency2-dstAccessMask-03089
  Any access flag included in dstAccessMask must be supported by one of the pipeline stages in dstStageMask, as specified in the table of supported access types

- VUID-VkSubpassDependency2-dependencyFlags-03090
  If dependencyFlags includes VK_DEPENDENCY_VIEW_LOCAL_BIT, srcSubpass must not be equal to VK_SUBPASS_EXTERNAL

- VUID-VkSubpassDependency2-dependencyFlags-03091
  If dependencyFlags includes VK_DEPENDENCY_VIEW_LOCAL_BIT, dstSubpass must not be equal to VK_SUBPASS_EXTERNAL

- VUID-VkSubpassDependency2-srcSubpass-02245
  If srcSubpass equals dstSubpass, and srcStageMask and dstStageMask both include a framebuffer-space stage, then dependencyFlags must include VK_DEPENDENCY_BY_REGION_BIT

- VUID-VkSubpassDependency2-viewOffset-02530
  If viewOffset is not equal to 0, srcSubpass must not be equal to dstSubpass

- VUID-VkSubpassDependency2-dependencyFlags-03092
If `dependencyFlags` does not include `VK_DEPENDENCY_VIEW_LOCAL_BIT`, `viewOffset` must be 0

- **VUID-VkSubpassDependency2-srcStageMask-02103**
  If the mesh shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`

- **VUID-VkSubpassDependency2-srcStageMask-02104**
  If the task shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

- **VUID-VkSubpassDependency2-dstStageMask-02105**
  If the mesh shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV`

- **VUID-VkSubpassDependency2-dstStageMask-02106**
  If the task shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV`

---

### Valid Usage (Implicit)

- **VUID-VkSubpassDependency2-sType-sType**
  sType must be `VK_STRUCTURE_TYPE_SUBPASS_DEPENDENCY_2`

- **VUID-VkSubpassDependency2-pNext-pNext**
  pNext must be NULL or a pointer to a valid instance of `VkMemoryBarrier2KHR`

- **VUID-VkSubpassDependency2-sType-unique**
  The sType value of each struct in the pNext chain must be unique

- **VUID-VkSubpassDependency2-srcStageMask-parameter**
  srcStageMask must be a valid combination of VkPipelineStageFlagBits values

- **VUID-VkSubpassDependency2-dstStageMask-parameter**
  dstStageMask must be a valid combination of VkPipelineStageFlagBits values

- **VUID-VkSubpassDependency2-srcAccessMask-parameter**
  srcAccessMask must be a valid combination of VkAccessFlagBits values

- **VUID-VkSubpassDependency2-dstAccessMask-parameter**
  dstAccessMask must be a valid combination of VkAccessFlagBits values

- **VUID-VkSubpassDependency2-dependencyFlags-parameter**
  dependencyFlags must be a valid combination of VkDependencyFlagBits values

---

To destroy a render pass, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyRenderPass(
    VkDevice    device,          // Provided by VK_VERSION_1_0
    VkRenderPass renderPass,     // Provided by VK_VERSION_1_0
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the render pass.
• `renderPass` is the handle of the render pass to destroy.

• `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyRenderPass-renderPass-00873**
  All submitted commands that refer to `renderPass` must have completed execution.

- **VUID-vkDestroyRenderPass-renderPass-00874**
  If `VkAllocationCallbacks` were provided when `renderPass` was created, a compatible set of callbacks must be provided here.

- **VUID-vkDestroyRenderPass-renderPass-00875**
  If no `VkAllocationCallbacks` were provided when `renderPass` was created, `pAllocator` must be `NULL`.

### Valid Usage (Implicit)

- **VUID-vkDestroyRenderPass-device-parameter**
  **device** must be a valid `VkDevice` handle.

- **VUID-vkDestroyRenderPass-renderPass-parameter**
  If `renderPass` is not `VK_NULL_HANDLE`, `renderPass` must be a valid `VkRenderPass` handle.

- **VUID-vkDestroyRenderPass-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- **VUID-vkDestroyRenderPass-renderPass-parent**
  If `renderPass` is a valid handle, it must have been created, allocated, or retrieved from `device`.

### Host Synchronization

- Host access to `renderPass` must be externally synchronized.

### 8.2. Render Pass Compatibility

Framebuffers and graphics pipelines are created based on a specific render pass object. They must only be used with that render pass object, or one compatible with it.

Two attachment references are compatible if they have matching format and sample count, or are both `VK_ATTACHMENT_UNUSED` or the pointer that would contain the reference is `NULL`.

Two arrays of attachment references are compatible if all corresponding pairs of attachments are compatible. If the arrays are of different lengths, attachment references not present in the smaller array are treated as `VK_ATTACHMENT_UNUSED`.

Chapter 8. Render Pass | 445
Two render passes are compatible if their corresponding color, input, resolve, and depth/stencil attachment references are compatible and if they are otherwise identical except for:

- Initial and final image layout in attachment descriptions
- Load and store operations in attachment descriptions
- Image layout in attachment references

As an additional special case, if two render passes have a single subpass, the resolve attachment reference and depth/stencil resolve mode compatibility requirements are ignored.

A framebuffer is compatible with a render pass if it was created using the same render pass or a compatible render pass.

### 8.3. Framebuffers

Render passes operate in conjunction with *framebuffers*. Framebuffers represent a collection of specific memory attachments that a render pass instance uses.

Framebuffers are represented by `VkFramebuffer` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFramebuffer)
```

To create a framebuffer, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateFramebuffer(
    VkDevice                                device,
    const VkFramebufferCreateInfo*         pCreateInfo,
    const VkAllocationCallbacks*           pAllocator,
    VkFramebuffer*                         pFramebuffer);
```

- `device` is the logical device that creates the framebuffer.
- `pCreateInfo` is a pointer to a `VkFramebufferCreateInfo` structure describing additional information about framebuffer creation.
- `pAllocator` controls host memory allocation as described in the *Memory Allocation* chapter.
- `pFramebuffer` is a pointer to a `VkFramebuffer` handle in which the resulting framebuffer object is returned.
Valid Usage

• VUID-vkCreateFramebuffer-pCreateInfo-02777
  If `pCreateInfo->flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, and `attachmentCount` is not 0, each element of `pCreateInfo->pAttachments` must have been created on device.

Valid Usage (Implicit)

• VUID-vkCreateFramebuffer-device-parameter
  `device` must be a valid `VkDevice` handle

• VUID-vkCreateFramebuffer-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkFramebufferCreateInfo` structure

• VUID-vkCreateFramebuffer-pAllocator-parameter
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

• VUID-vkCreateFramebuffer-pFramebuffer-parameter
  `pFramebuffer` must be a valid pointer to a `VkFramebuffer` handle

Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkFramebufferCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkFramebufferCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkFramebufferCreateFlags flags;
  VkRenderPass renderPass;
  uint32_t attachmentCount;
  const VkImageView* pAttachments;
  uint32_t width;
  uint32_t height;
  uint32_t layers;
} VkFramebufferCreateInfo;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **flags** is a bitmask of **VkFramebufferCreateFlagBits**

• **renderPass** is a render pass defining what render passes the framebuffer will be compatible with. See **Render Pass Compatibility** for details.

• **attachmentCount** is the number of attachments.

• **pAttachments** is a pointer to an array of **VkImageView** handles, each of which will be used as the corresponding attachment in a render pass instance. If **flags** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, this parameter is ignored.

• **width**, **height** and **layers** define the dimensions of the framebuffer. If the render pass uses multiview, then **layers must** be one and each attachment requires a number of layers that is greater than the maximum bit index set in the view mask in the subpasses in which it is used.

Other than the exceptions listed below, applications **must** ensure that all accesses to memory that backs image subresources used as attachments in a given renderpass instance either happen-before the **load operations** for those attachments, or happen-after the **store operations** for those attachments.

The exceptions to the general rule are:

• For depth/stencil attachments, an aspect **can** be used separately as attachment and non-attachment if both accesses are read-only.

• For depth/stencil attachments, each aspect **can** be used separately as attachments and non-attachments as long as the non-attachment accesses are also via an image subresource in either the **VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL** layout or the **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL** layout, and the attachment resource uses whichever of those two layouts the image accesses do not.

Use of non-attachment aspects in these cases is only well defined if the attachment is used in the subpass where the non-attachment access is being made, or the layout of the image subresource is constant throughout the entire render pass instance, including the **initialLayout** and **finalLayout**.

**Note**

These restrictions mean that the render pass has full knowledge of all uses of all of the attachments, so that the implementation is able to make correct decisions about when and how to perform layout transitions, when to overlap execution of subpasses, etc.

It is legal for a subpass to use no color or depth/stencil attachments, either because it has no attachment references or because all of them are **VK_ATTACHMENT_UNUSED**. This kind of subpass **can** use shader side effects such as image stores and atomics to produce an output. In this case, the subpass continues to use the **width**, **height**, and **layers** of the framebuffer to define the dimensions of the rendering area, and the **rasterizationSamples** from each pipeline's **VkPipelineMultisampleStateCreateInfo** to define the number of samples used in rasterization; however, if **VkPhysicalDeviceFeatures::variableMultisampleRate** is **VK_FALSE**, then all pipelines to be bound with the subpass **must** have the same value for **VkPipelineMultisampleStateCreateInfo**.
::rasterizationSamples.
Valid Usage

- VUID-VkFramebufferCreateInfo-attachmentCount-00876
  `attachmentCount` must be equal to the attachment count specified in `renderPass`

- VUID-VkFramebufferCreateInfo-flags-02778
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, and `attachmentCount` is not 0, `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkImageView` handles

- VUID-VkFramebufferCreateInfo-pAttachments-00877
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` that is used as a color attachment or resolve attachment by `renderPass` must have been created with a `usage` value including `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`

- VUID-VkFramebufferCreateInfo-pAttachments-02633
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` that is used as a depth/stencil attachment by `renderPass` must have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

- VUID-VkFramebufferCreateInfo-pAttachments-02634
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` that is used as a depth/stencil resolve attachment by `renderPass` must have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

- VUID-VkFramebufferCreateInfo-pAttachments-00879
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` that is used as an input attachment by `renderPass` must have been created with a `usage` value including `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

- VUID-VkFramebufferCreateInfo-pAttachments-02552
  Each element of `pAttachments` that is used as a fragment density map attachment by `renderPass` must not have been created with a `flags` value including `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

- VUID-VkFramebufferCreateInfo-renderPass-02553
  If `renderPass` has a fragment density map attachment and non-subsample image feature is not enabled, each element of `pAttachments` must have been created with a `flags` value including `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT` unless that element is the fragment density map attachment

- VUID-VkFramebufferCreateInfo-pAttachments-00880
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must have been created with a `VkFormat` value that matches the `VkFormat` specified by the corresponding `VkAttachmentDescription` in `renderPass`

- VUID-VkFramebufferCreateInfo-pAttachments-00881
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must have been created with a `samples` value that matches the `samples` value specified by the corresponding `VkAttachmentDescription` in `renderPass`

- VUID-VkFramebufferCreateInfo-flags-04533
  If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of
pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have been created with a VkImageCreateInfo::width greater than or equal to width

• VUID-VkFramebufferCreateInfo-flags-04534
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have been created with a VkImageCreateInfo::height greater than or equal to height

• VUID-VkFramebufferCreateInfo-flags-04535
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have been created with a VkImageViewCreateInfo::subresourceRange.layerCount greater than or equal to layers

• VUID-VkFramebufferCreateInfo-renderPass-04536
  If renderPass was specified with non-zero view masks, each element of pAttachments that is used as an input, color, resolve, or depth/stencil attachment by renderPass must have a layerCount greater than the index of the most significant bit set in any of those view masks

• VUID-VkFramebufferCreateInfo-renderPass-02746
  If renderPass was specified with non-zero view masks, each element of pAttachments that is referenced by fragmentDensityMapAttachment must have a layerCount equal to 1 or greater than the index of the most significant bit set in any of those view masks

• VUID-VkFramebufferCreateInfo-renderPass-02747
  If renderPass was not specified with non-zero view masks, each element of pAttachments that is referenced by fragmentDensityMapAttachment must have a layerCount equal to 1

• VUID-VkFramebufferCreateInfo-pAttachments-02555
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, an element of pAttachments that is referenced by fragmentDensityMapAttachment must have a width at least as large as \[
  \left\lceil \frac{\text{maxFragmentDensityPerPixelSize.width}}{\text{width}} \right\rceil
  \]

• VUID-VkFramebufferCreateInfo-pAttachments-02556
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, an element of pAttachments that is referenced by fragmentDensityMapAttachment must have a height at least as large as \[
  \left\lceil \frac{\text{maxFragmentDensityPerPixelSize.height}}{\text{height}} \right\rceil
  \]

• VUID-VkFramebufferCreateInfo-flags-04537
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, and renderPass was specified with non-zero view masks, each element of pAttachments that is used as a fragment shading rate attachment by renderPass must have a layerCount that is either 1, or greater than the index of the most significant bit set in any of those view masks

• VUID-VkFramebufferCreateInfo-flags-04538
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, and renderPass was not specified with non-zero view masks, each element of pAttachments that is used as a fragment shading rate attachment by renderPass must have a layerCount that is either 1, or greater than layers

• VUID-VkFramebufferCreateInfo-flags-04539
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, an element of pAttachments
that is used as a **fragment shading rate attachment** must have a width at least as large as $\lceil \text{width} / \text{texelWidth} \rceil$, where \text{texelWidth} is the largest value of shadingRateAttachmentTexelSize.width in a \text{VkFragmentShadingRateAttachmentInfoKHR} which references that attachment.

- **VUID-VkFramebufferCreateInfo-flags-04540**
  
  If \text{flags} does not include \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, an element of \text{pAttachments} that is used as a **fragment shading rate attachment** must have a height at least as large as $\lceil \text{height} / \text{texelHeight} \rceil$, where \text{texelHeight} is the largest value of shadingRateAttachmentTexelSize.height in a \text{VkFragmentShadingRateAttachmentInfoKHR} which references that attachment.

- **VUID-VkFramebufferCreateInfo-pAttachments-00883**
  
  If \text{flags} does not include \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, each element of \text{pAttachments} must only specify a single mip level.

- **VUID-VkFramebufferCreateInfo-pAttachments-00884**
  
  If \text{flags} does not include \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, each element of \text{pAttachments} must have been created with the identity swizzle.

- **VUID-VkFramebufferCreateInfo-width-00885**
  
  \text{width} must be greater than 0.

- **VUID-VkFramebufferCreateInfo-width-00886**
  
  \text{width} must be less than or equal to \text{VkPhysicalDeviceLimits::maxFramebufferWidth}.

- **VUID-VkFramebufferCreateInfo-height-00887**
  
  \text{height} must be greater than 0.

- **VUID-VkFramebufferCreateInfo-height-00888**
  
  \text{height} must be less than or equal to \text{VkPhysicalDeviceLimits::maxFramebufferHeight}.

- **VUID-VkFramebufferCreateInfo-layers-00889**
  
  \text{layers} must be greater than 0.

- **VUID-VkFramebufferCreateInfo-layers-00890**
  
  \text{layers} must be less than or equal to \text{VkPhysicalDeviceLimits::maxFramebufferLayers}.

- **VUID-VkFramebufferCreateInfo-renderPass-02531**
  
  If \text{renderPass} was specified with non-zero view masks, \text{layers} must be 1.

- **VUID-VkFramebufferCreateInfo-pAttachments-00891**
  
  If \text{flags} does not include \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, each element of \text{pAttachments} that is a 2D or 2D array image view taken from a 3D image must not be a depth/stencil format.

- **VUID-VkFramebufferCreateInfo-flags-03189**
  
  If the \text{imageless framebuffer} feature is not enabled, \text{flags} must not include \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}.

- **VUID-VkFramebufferCreateInfo-flags-03190**
  
  If \text{flags} includes \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the \text{pNext} chain must include an instance of \text{VkFramebufferAttachmentsCreateInfo}.

- **VUID-VkFramebufferCreateInfo-flags-03191**
  
  If \text{flags} includes \text{VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT}, the attachmentImageInfoCount member of an instance of \text{VkFramebufferAttachmentsCreateInfo} in the \text{pNext} chain must
be equal to either zero or attachmentCount

• VUID-VkFramebufferCreateInfo-flags-04541

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the width member of any element of the pAttachmentImageInfos member of an instance of VkFramebufferAttachmentsCreateInfo in the pNext chain that is used as an input, color, resolve or depth/stencil attachment in pRenderPass must be greater than or equal to width

• VUID-VkFramebufferCreateInfo-flags-04542

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the height member of any element of the pAttachmentImageInfos member of an instance of VkFramebufferAttachmentsCreateInfo in the pNext chain that is used as an input, color, resolve or depth/stencil attachment in pRenderPass must be greater than or equal to height

• VUID-VkFramebufferCreateInfo-flags-03196

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the width member of any element of the pAttachmentImageInfos member of an instance of VkFramebufferAttachmentsCreateInfo in the pNext chain that is referenced by VkRenderPassFragmentDensityMapCreateInfoEXT::fragmentDensityMapAttachment in renderPass must be greater than or equal to \( \left\lceil \frac{width}{\max\text{FragmentDensityTexelSize.width}} \right\rceil \)

• VUID-VkFramebufferCreateInfo-flags-03197

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the height member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain that is referenced by VkRenderPassFragmentDensityMapCreateInfoEXT::fragmentDensityMapAttachment in renderPass must be greater than or equal to \( \left\lceil \frac{height}{\max\text{FragmentDensityTexelSize.height}} \right\rceil \)

• VUID-VkFramebufferCreateInfo-flags-04543

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the width member of any element of the pAttachmentImageInfos member of an instance of VkFramebufferAttachmentsCreateInfo in the pNext chain that is used as a fragment shading rate attachment must be greater than or equal to \( \left\lceil \frac{width}{\text{texelWidth}} \right\rceil \), where texelWidth is the largest value of shadingRateAttachmentTexelSize.width in a VkFragmentShadingRateAttachmentInfoKHR which references that attachment

• VUID-VkFramebufferCreateInfo-flags-04544

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the height member of any element of the pAttachmentImageInfos member of an instance of VkFramebufferAttachmentsCreateInfo in the pNext chain that is used as a fragment shading rate attachment must be greater than or equal to \( \left\lceil \frac{height}{\text{texelHeight}} \right\rceil \), where texelHeight is the largest value of shadingRateAttachmentTexelSize.height in a VkFragmentShadingRateAttachmentInfoKHR which references that attachment

• VUID-VkFramebufferCreateInfo-flags-04545

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the layerCount member of any element of the pAttachmentImageInfos member of an instance of VkFramebufferAttachmentsCreateInfo in the pNext chain that is used as a fragment shading rate attachment must be either 1, or greater than or equal to layers

• VUID-VkFramebufferCreateInfo-flags-04587

If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, and renderPass was specified with non-zero view masks, each element of pAttachments that is used as a fragment shading
rate attachment by renderPass must have a layerCount that is either 1, or greater than the index of the most significant bit set in any of those view masks

- VUID-VkFramebufferCreateInfo-renderPass-03198
  If multiview is enabled for renderPass, and flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the layerCount member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain used as an input, color, resolve, or depth/stencil attachment in pRenderPass must be greater than the maximum bit index set in the view mask in the subpasses in which it is used in renderPass

- VUID-VkFramebufferCreateInfo-renderPass-04546
  If multiview is not enabled for renderPass, and flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the layerCount member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain used as an input, color, resolve, or depth/stencil attachment in pRenderPass must be greater than or equal to layers

- VUID-VkFramebufferCreateInfo-flags-03201
  If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the usage member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain that refers to an attachment used as a color attachment or resolve attachment by renderPass must include VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT

- VUID-VkFramebufferCreateInfo-flags-03202
  If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the usage member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain that refers to an attachment used as a depth/stencil attachment by renderPass must include VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

- VUID-VkFramebufferCreateInfo-flags-03203
  If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the usage member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain that refers to an attachment used as a depth/stencil resolve attachment by renderPass must include VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT

- VUID-VkFramebufferCreateInfo-flags-03204
  If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the usage member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain that refers to an attachment used as an input attachment by renderPass must include VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

- VUID-VkFramebufferCreateInfo-flags-03205
  If flags includes VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, at least one element of the pViewFormats member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain must be equal to the corresponding value of VkAttachmentDescription::format used to create renderPass

- VUID-VkFramebufferCreateInfo-flags-04113
  If flags does not include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of
pAttachments ***must*** have been created with **VkImageViewCreateInfo::viewType** not equal to **VK_IMAGE_VIEW_TYPE_3D**

- **VUID-VkFramebufferCreateInfo-flags-04548**
  - If ***flags*** does not include **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, each element of pAttachments that is used as a fragment shading rate attachment by renderPass ***must*** have been created with a ***usage*** value including **VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**

- **VUID-VkFramebufferCreateInfo-flags-04549**
  - If ***flags*** includes **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT**, the ***usage*** member of any element of the pAttachmentImageInfos member of a VkFramebufferAttachmentsCreateInfo structure included in the pNext chain that refers to an attachment used as a fragment shading rate attachment by renderPass ***must*** include **VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR**

### Valid Usage (Implicit)

- **VUID-VkFramebufferCreateInfo-sType-sType**
  - sType ***must*** be **VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO**

- **VUID-VkFramebufferCreateInfo-pNext-pNext**
  - pNext ***must*** be NULL or a pointer to a valid instance of VkFramebufferAttachmentsCreateInfo

- **VUID-VkFramebufferCreateInfo-sType-unique**
  - The sType value of each struct in the pNext chain ***must*** be unique

- **VUID-VkFramebufferCreateInfo-flags-parameter**
  - flags ***must*** be a valid combination of VkFramebufferCreateFlagBits values

- **VUID-VkFramebufferCreateInfo-renderPass-parameter**
  - renderPass ***must*** be a valid VkRenderPass handle

- **VUID-VkFramebufferCreateInfo-commonparent**
  - Both of renderPass, and the elements of pAttachments that are valid handles of non-ignored parameters ***must*** have been created, allocated, or retrieved from the same VkDevice

The VkFramebufferAttachmentsCreateInfo structure is defined as:

```c
typedef struct VkFramebufferAttachmentsCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    uint32_t                                    attachmentImageInfoCount;
    const VkFramebufferAttachmentImageInfo*    pAttachmentImageInfos;
} VkFramebufferAttachmentsCreateInfo;
```

or the equivalent
Provided by VK_KHR_imageless_framebuffer
typedef VkFramebufferAttachmentsCreateInfo VkFramebufferAttachmentsCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **attachmentImageInfoCount** is the number of attachments being described.
- **pAttachmentImageInfos** is a pointer to an array of VkFramebufferAttachmentImageInfo instances, each of which describes a number of parameters of the corresponding attachment in a render pass instance.

Valid Usage (Implicit)

- **VUID-VkFramebufferAttachmentsCreateInfo-sType-sType**
  - **sType** must be VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO
- **VUID-VkFramebufferAttachmentsCreateInfo-pAttachmentImageInfos-parameter**
  - If attachmentImageInfoCount is not 0, pAttachmentImageInfos must be a valid pointer to an array of attachmentImageInfoCount valid VkFramebufferAttachmentImageInfo structures

The VkFramebufferAttachmentImageInfo structure is defined as:

```c
typedef struct VkFramebufferAttachmentImageInfo {
    VkStructureType           sType;
    const void*               pNext;
    VkImageCreateFlags        flags;
    VkImageUsageFlags         usage;
    uint32_t                   width;
    uint32_t                   height;
    uint32_t                   layerCount;
    uint32_t                   viewFormatCount;
    const VkFormat*           pViewFormats;
} VkFramebufferAttachmentImageInfo;
```

or the equivalent

Provided by VK_KHR_imageless_framebuffer
typedef VkFramebufferAttachmentImageInfo VkFramebufferAttachmentImageInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of VkImageCreateFlagBits, matching the value of VkImageCreateInfo::flags used to create an image that will be used with this framebuffer.
- **usage** is a bitmask of VkImageUsageFlagBits, matching the value of VkImageCreateInfo::usage

456 | Chapter 8. Render Pass
used to create an image used with this framebuffer.

- **width** is the width of the image view used for rendering.
- **height** is the height of the image view used for rendering.
- **viewFormatCount** is the number of entries in the `pViewFormats` array, matching the value of `VkImageFormatListCreateInfo::viewFormatCount` used to create an image used with this framebuffer.
- **pViewFormats** is an array which lists all formats which can be used when creating views of the image, matching the value of `VkImageFormatListCreateInfo::pViewFormats` used to create an image used with this framebuffer.

Images that can be used with the framebuffer when beginning a render pass, as specified by `VkRenderPassAttachmentBeginInfo`, **must** be created with parameters that are identical to those specified here.

### Valid Usage (Implicit)

- VUID-VkFramebufferAttachmentImageInfo-sType-sType
  
  sType **must** be `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO`

- VUID-VkFramebufferAttachmentImageInfo-pNext-pNext
  
  pNext **must** be `NULL`

- VUID-VkFramebufferAttachmentImageInfo-flags-parameter
  
  flags **must** be a valid combination of `VkImageCreateFlagBits` values

- VUID-VkFramebufferAttachmentImageInfo-usage-parameter
  
  usage **must** be a valid combination of `VkImageUsageFlagBits` values

- VUID-VkFramebufferAttachmentImageInfo-usage-required bitmask
  
  usage **must** not be `0`

- VUID-VkFramebufferAttachmentImageInfo-pViewFormats-parameter
  
  If `viewFormatCount` is not `0`, `pViewFormats **must** be a valid pointer to an array of `viewFormatCount` valid `VkFormat` values

Bits which can be set in `VkFramebufferCreateInfo::flags` to specify options for framebuffers are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkFramebufferCreateFlagBits {
    VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT = 0x00000001,
    // Provided by VK_KHR_imageless_framebuffer
    VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT_KHR = VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT,
} VkFramebufferCreateFlagBits;
```

- **VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT** specifies that image views are not specified, and only attachment compatibility information will be provided via a `VkFramebufferAttachmentImageInfo` structure.
typedef VkFlagsVkFramebufferCreateFlags;

VkFramebufferCreateFlags is a bitmask type for setting a mask of zero or more VkFramebufferCreateFlagBits.

To destroy a framebuffer, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyFramebuffer(
    VkDevice device,
    VkFramebuffer framebuffer,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the framebuffer.
- **framebuffer** is the handle of the framebuffer to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyFramebuffer-framebuffer-00892**
  All submitted commands that refer to framebuffer **must** have completed execution

- **VUID-vkDestroyFramebuffer-framebuffer-00893**
  If VkAllocationCallbacks were provided when framebuffer was created, a compatible set of callbacks **must** be provided here

- **VUID-vkDestroyFramebuffer-framebuffer-00894**
  If no VkAllocationCallbacks were provided when framebuffer was created, pAllocator **must** be NULL

### Valid Usage (Implicit)

- **VUID-vkDestroyFramebuffer-device-parameter**
  device **must** be a valid VkDevice handle

- **VUID-vkDestroyFramebuffer-framebuffer-parameter**
  If framebuffer is not VK_NULL_HANDLE, framebuffer **must** be a valid VkFramebuffer handle

- **VUID-vkDestroyFramebuffer-pAllocator-parameter**
  If pAllocator is not NULL, pAllocator **must** be a valid pointer to a valid VkAllocationCallbacks structure

- **VUID-vkDestroyFramebuffer-framebuffer-parent**
  If framebuffer is a valid handle, it **must** have been created, allocated, or retrieved from device
8.4. Render Pass Commands

An application records the commands for a render pass instance one subpass at a time, by beginning a render pass instance, iterating over the subpasses to record commands for that subpass, and then ending the render pass instance.

To begin a render pass instance, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBeginRenderPass(
    VkCommandBuffer                             commandBuffer,
    const VkRenderPassBeginInfo*                pRenderPassBegin,
    VkSubpassContents                           contents);
```

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a `VkRenderPassBeginInfo` structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.
- `contents` is a `VkSubpassContents` value specifying how the commands in the first subpass will be provided.

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.
Valid Usage

- **VUID-vkCmdBeginRenderPass-initialLayout-00895**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`

- **VUID-vkCmdBeginRenderPass-initialLayout-01758**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

- **VUID-vkCmdBeginRenderPass-initialLayout-02842**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

- **VUID-vkCmdBeginRenderPass-stencilInitialLayout-02843**
  If any of the `stencilInitialLayout` or `stencilFinalLayout` member of the `VkAttachmentDescriptionStencilLayout` structures or the `stencilLayout` member of the `VkAttachmentReferenceStencilLayout` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

- **VUID-vkCmdBeginRenderPass-initialLayout-00897**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have
been created with a usage value including `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

- VUID-vkCmdBeginRenderPass-initialLayout-00898
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`

- VUID-vkCmdBeginRenderPass-initialLayout-00899
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including `VK_IMAGE_USAGE_TRANSFER_DST_BIT`

- VUID-vkCmdBeginRenderPass-initialLayout-00900
  If any of the initialLayout members of the VkAttachmentDescription structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is not `VK_IMAGE_LAYOUT_UNDEFINED`, then each such initialLayout must be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the framebuffer member of pRenderPassBegin

- VUID-vkCmdBeginRenderPass-srcStageMask-00901
  The srcStageMask and dstStageMask members of any element of the pDependencies member of VkRenderPassCreateInfo used to create renderPass must be supported by the capabilities of the queue family identified by the queueFamilyIndex member of the VkCommandPoolCreateInfo used to create the command pool which commandBuffer was allocated from

- VUID-vkCmdBeginRenderPass-framebuffer-02532
  For any attachment in framebuffer that is used by renderPass and is bound to memory locations that are also bound to another attachment used by renderPass, and if at least one of those uses causes either attachment to be written to, both attachments must have had the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` set
Valid Usage (Implicit)

- VUID-vkCmdBeginRenderPass-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdBeginRenderPass-pRenderPassBegin-parameter
  
  `pRenderPassBegin` must be a valid pointer to a valid `VkRenderPassBeginInfo` structure

- VUID-vkCmdBeginRenderPass-contents-parameter
  
  `contents` must be a valid `VkSubpassContents` value

- VUID-vkCmdBeginRenderPass-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdBeginRenderPass-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdBeginRenderPass-renderpass
  
  This command must only be called outside of a render pass instance

- VUID-vkCmdBeginRenderPass-bufferlevel
  
  `commandBuffer` must be a primary `VkCommandBuffer`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

Alternatively to begin a render pass, call:

```c
void vkCmdBeginRenderPass2(
    VkCommandBuffer          commandBuffer,
    const VkRenderPassBeginInfo* pRenderPassBegin,
    const VkSubpassBeginInfo*  pSubpassBeginInfo);
```
void vkCmdBeginRenderPass2KHR(
    VkCommandBuffer commandBuffer,
    const VkRenderPassBeginInfo* pRenderPassBegin,
    const VkSubpassBeginInfo* pSubpassBeginInfo);

- `commandBuffer` is the command buffer in which to record the command.
- `pRenderPassBegin` is a pointer to a `VkRenderPassBeginInfo` structure specifying the render pass to begin an instance of, and the framebuffer the instance uses.
- `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfo` structure containing information about the subpass which is about to begin rendering.

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.
Valid Usage

- **VUID-vkCmdBeginRenderPass2-framebuffer-02779**
  Both the framebuffer and renderPass members of pRenderPassBegin must have been created on the same VkDevice that commandBuffer was allocated on.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03094**
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03096**
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

- **VUID-vkCmdBeginRenderPass2-initialLayout-02844**
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

- **VUID-vkCmdBeginRenderPass2-stencilInitialLayout-02845**
  If any of the stencilInitialLayout or stencilFinalLayout member of the VkAttachmentDescriptionStencilLayout structures or the stencilLayout member of the VkAttachmentReferenceStencilLayout structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03097**
  If any of the initialLayout or finalLayout member of the VkAttachmentDescription structures or the layout member of the VkAttachmentReference structures specified when creating the render pass specified in the renderPass member of pRenderPassBegin is VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL then the corresponding attachment image view of the framebuffer specified in the framebuffer member of pRenderPassBegin must have been created with a usage value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.
creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_SAMPLED_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03098**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_SRC_BIT`.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03099**
  If any of the `initialLayout` or `finalLayout` member of the `VkAttachmentDescription` structures or the `layout` member of the `VkAttachmentReference` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` then the corresponding attachment image view of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin` must have been created with a `usage` value including `VK_IMAGE_USAGE_TRANSFER_DST_BIT`.

- **VUID-vkCmdBeginRenderPass2-initialLayout-03100**
  If any of the `initialLayout` members of the `VkAttachmentDescription` structures specified when creating the render pass specified in the `renderPass` member of `pRenderPassBegin` is not `VK_IMAGE_LAYOUT_UNDEFINED`, then each such `initialLayout` must be equal to the current layout of the corresponding attachment image subresource of the framebuffer specified in the `framebuffer` member of `pRenderPassBegin`.

- **VUID-vkCmdBeginRenderPass2-srcStageMask-03101**
  The `srcStageMask` and `dstStageMask` members of any element of the `pDependencies` member of `VkRenderPassCreateInfo` used to create `renderPass` must be supported by the capabilities of the queue family identified by the `queueFamilyIndex` member of the `VkCommandPoolCreateInfo` used to create the command pool which `commandBuffer` was allocated from.

- **VUID-vkCmdBeginRenderPass2-framebuffer-02533**
  For any attachment in `framebuffer` that is used by `renderPass` and is bound to memory locations that are also bound to another attachment used by `renderPass`, and if at least one of those uses causes either attachment to be written to, both attachments must have had the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT` set.
Valid Usage (Implicit)

- **VUID-vkCmdBeginRenderPass2-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdBeginRenderPass2-pRenderPassBegin-parameter**
  
  `pRenderPassBegin` must be a valid pointer to a valid `VkRenderPassBeginInfo` structure.

- **VUID-vkCmdBeginRenderPass2-pSubpassBeginInfo-parameter**
  
  `pSubpassBeginInfo` must be a valid pointer to a valid `VkSubpassBeginInfo` structure.

- **VUID-vkCmdBeginRenderPass2-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdBeginRenderPass2-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- **VUID-vkCmdBeginRenderPass2-renderpass**
  
  This command must only be called outside of a render pass instance.

- **VUID-vkCmdBeginRenderPass2-bufferlevel**
  
  `commandBuffer` must be a primary `VkCommandBuffer`.

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

The `VkRenderPassBeginInfo` structure is defined as:
typedef struct VkRenderPassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkRenderPass renderPass;
    VkFramebuffer framebuffer;
    VkRect2D renderArea;
    uint32_t clearValueCount;
    const VkClearValue* pClearValues;
} VkRenderPassBeginInfo;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **renderPass** is the render pass to begin an instance of.
- **framebuffer** is the framebuffer containing the attachments that are used with the render pass.
- **renderArea** is the render area that is affected by the render pass instance, and is described in more detail below.
- **clearValueCount** is the number of elements in **pClearValues**.
- **pClearValues** is a pointer to an array of **clearValueCount** VkClearValue structures that contains clear values for each attachment, if the attachment uses a **loadOp** value of VK_ATTACHMENT_LOAD_OP_CLEAR or if the attachment has a depth/stencil format and uses a **stencilLoadOp** value of VK_ATTACHMENT_LOAD_OP_CLEAR. The array is indexed by attachment number. Only elements corresponding to cleared attachments are used. Other elements of **pClearValues** are ignored.

**renderArea** is the render area that is affected by the render pass instance. The effects of attachment load, store and multisample resolve operations are restricted to the pixels whose x and y coordinates fall within the render area on all attachments. The render area extends to all layers of framebuffer. The application **must** ensure (using scissor if necessary) that all rendering is contained within the render area. The render area, after any transform specified by VkRenderPassTransformBeginInfoQCOM::transform is applied, **must** be contained within the framebuffer dimensions.

If **render pass transform** is enabled, then **renderArea** **must** equal the framebuffer pre-transformed dimensions. After **renderArea** has been transformed by VkRenderPassTransformBeginInfoQCOM::transform, the resulting render area **must** be equal to the framebuffer dimensions.

When multiview is enabled, the resolve operation at the end of a subpass applies to all views in the view mask.

**Note**

There **may** be a performance cost for using a render area smaller than the framebuffer, unless it matches the render area granularity for the render pass.
Valid Usage

- **VUID-VkRenderPassBeginInfo-clearValueCount-00902**
  
  `clearValueCount` must be greater than the largest attachment index in `renderPass` that specifies a `loadOp` (or `stencilLoadOp`, if the attachment has a depth/stencil format) of `VK_ATTACHMENT_LOAD_OP_CLEAR`

- **VUID-VkRenderPassBeginInfo-renderPass-00904**
  
  `renderPass` must be compatible with the `renderPass` member of the `VkFramebufferCreateInfo` structure specified when creating `framebuffer`

- **VUID-VkRenderPassBeginInfo-pNext-02850**
  
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.x` must be greater than or equal to 0

- **VUID-VkRenderPassBeginInfo-pNext-02851**
  
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.y` must be greater than or equal to 0

- **VUID-VkRenderPassBeginInfo-pNext-02852**
  
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.width` must be less than or equal to `VkFramebufferCreateInfo::width` the framebuffer was created with

- **VUID-VkRenderPassBeginInfo-pNext-02853**
  
  If the `pNext` chain does not contain `VkDeviceGroupRenderPassBeginInfo` or its `deviceRenderAreaCount` member is equal to 0, `renderArea.offset.height` must be less than or equal to `VkFramebufferCreateInfo::height` the framebuffer was created with

- **VUID-VkRenderPassBeginInfo-pNext-02854**
  
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, the `offset.x` member of each element of `pDeviceRenderAreas` must be greater than or equal to 0

- **VUID-VkRenderPassBeginInfo-pNext-02855**
  
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, the `offset.y` member of each element of `pDeviceRenderAreas` must be greater than or equal to 0

- **VUID-VkRenderPassBeginInfo-pNext-02856**
  
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, `offset.x + offset.width` of each element of `pDeviceRenderAreas` must be less than or equal to `VkFramebufferCreateInfo::width` the framebuffer was created with

- **VUID-VkRenderPassBeginInfo-pNext-02857**
  
  If the `pNext` chain contains `VkDeviceGroupRenderPassBeginInfo`, `offset.y + offset.height` of each element of `pDeviceRenderAreas` must be less than or equal to `VkFramebufferCreateInfo::height` the framebuffer was created with

- **VUID-VkRenderPassBeginInfo-framebuffer-03207**
  
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that did not
include VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, and the pNext chain includes a VkRenderPassAttachmentBeginInfo structure, its attachmentCount must be zero.

- VUID-VkRenderPassBeginInfo-framebuffer-03208
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, the attachmentCount of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be equal to the value of VkFramebufferAttachmentsCreateInfo::attachmentImageInfoCount used to create framebuffer.

- VUID-VkRenderPassBeginInfo-framebuffer-02780
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must have been created on the same VkDevice as framebuffer and renderPass.

- VUID-VkRenderPassBeginInfo-framebuffer-03209
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView of an image created with a value of VkImageCreateInfo::flags equal to the flags member of the corresponding element of VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer.

- VUID-VkRenderPassBeginInfo-framebuffer-04627
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView with an inherited usage equal to the usage member of the corresponding element of VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer.

- VUID-VkRenderPassBeginInfo-framebuffer-03211
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView with a width equal to the width member of the corresponding element of VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer.

- VUID-VkRenderPassBeginInfo-framebuffer-03212
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView with a height equal to the height member of the corresponding element of VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer.

- VUID-VkRenderPassBeginInfo-framebuffer-03213
  If framebuffer was created with a VkFramebufferCreateInfo::flags value that included VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT, each element of the pAttachments member of a VkRenderPassAttachmentBeginInfo structure included in the pNext chain must be a VkImageView of an image created with a value of VkImageViewCreateInfo::subresourceRange.layerCount equal to the layerCount member of the corresponding element of VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos used to create framebuffer.
element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create framebuffer

- VUID-VkRenderPassBeginInfo-framebuffer-03214
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a value of `VkImageFormatListCreateInfo::viewFormatCount` equal to the `viewFormatCount` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create framebuffer

- VUID-VkRenderPassBeginInfo-framebuffer-03215
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a set of elements in `VkImageFormatListCreateInfo::pViewFormats` equal to the set of elements in the `pViewFormats` member of the corresponding element of `VkFramebufferAttachmentsCreateInfo::pAttachmentImageInfos` used to create framebuffer

- VUID-VkRenderPassBeginInfo-framebuffer-03216
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a value of `VkImageViewCreateInfo::format` equal to the corresponding value of `VkAttachmentDescription::format` in `renderPass`

- VUID-VkRenderPassBeginInfo-framebuffer-03217
  If `framebuffer` was created with a `VkFramebufferCreateInfo::flags` value that included `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of the `pAttachments` member of a `VkRenderPassAttachmentBeginInfo` structure included in the `pNext` chain must be a `VkImageView` of an image created with a value of `VkImageCreateInfo::samples` equal to the corresponding value of `VkAttachmentDescription::samples` in `renderPass`

- VUID-VkRenderPassBeginInfo-pNext-02869
  If the `pNext` chain includes `VkRenderPassTransformBeginInfoQCOM`, `renderArea.offset` must equal (0,0)

- VUID-VkRenderPassBeginInfo-pNext-02870
  If the `pNext` chain includes `VkRenderPassTransformBeginInfoQCOM`, `renderArea.extent` transformed by `VkRenderPassTransformBeginInfoQCOM::transform` must equal the framebuffer dimensions
Valid Usage (Implicit)

- VUID-VkRenderPassBeginInfo-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO

- VUID-VkRenderPassBeginInfo-pNext-pNext
  
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkDeviceGroupRenderPassBeginInfo, VkRenderPassAttachmentBeginInfo, VkRenderPassSampleLocationsBeginInfoEXT, or VkRenderPassTransformBeginInfoQCOM

- VUID-VkRenderPassBeginInfo-sType-unique
  
  The sType value of each struct in the pNext chain must be unique

- VUID-VkRenderPassBeginInfo-renderPass-parameter
  
  renderPass must be a valid VkRenderPass handle

- VUID-VkRenderPassBeginInfo-framebuffer-parameter
  
  framebuffer must be a valid VkFramebuffer handle

- VUID-VkRenderPassBeginInfo-commonparent
  
  Both of framebuffer, and renderPass must have been created, allocated, or retrieved from the same VkDevice

The image layout of the depth aspect of a depth/stencil attachment referring to an image created with VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT is dependent on the last sample locations used to render to the image subresource, thus preserving the contents of such depth/stencil attachments across subpass boundaries requires the application to specify these sample locations whenever a layout transition of the attachment may occur. This information can be provided by adding a VkRenderPassSampleLocationsBeginInfoEXT structure to the pNext chain of VkRenderPassBeginInfo.

The VkRenderPassSampleLocationsBeginInfoEXT structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkRenderPassSampleLocationsBeginInfoEXT {
    VkStructureType                          sType;
    const void*                              pNext;
    uint32_t                                 attachmentInitialSampleLocationsCount;
    const VkAttachmentSampleLocationsEXT*    pAttachmentInitialSampleLocations;
    uint32_t                                 postSubpassSampleLocationsCount;
    const VkSubpassSampleLocationsEXT*       pPostSubpassSampleLocations;
} VkRenderPassSampleLocationsBeginInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- attachmentInitialSampleLocationsCount is the number of elements in the pAttachmentInitialSampleLocations array.
• **pAttachmentInitialSampleLocations** is a pointer to an array of `attachmentInitialSampleLocationsCount` `VkAttachmentSampleLocationsEXT` structures specifying the attachment indices and their corresponding sample location state. Each element of `pAttachmentInitialSampleLocations` can specify the sample location state to use in the automatic layout transition performed to transition a depth/stencil attachment from the initial layout of the attachment to the image layout specified for the attachment in the first subpass using it.

• **postSubpassSampleLocationsCount** is the number of elements in the `pPostSubpassSampleLocations` array.

• **pPostSubpassSampleLocations** is a pointer to an array of `postSubpassSampleLocationsCount` `VkSubpassSampleLocationsEXT` structures specifying the subpass indices and their corresponding sample location state. Each element of `pPostSubpassSampleLocations` can specify the sample location state to use in the automatic layout transition performed to transition the depth/stencil attachment used by the specified subpass to the image layout specified in a dependent subpass or to the final layout of the attachment in case the specified subpass is the last subpass using that attachment. In addition, if `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE`, each element of `pPostSubpassSampleLocations` must specify the sample location state that matches the sample locations used by all pipelines that will be bound to a command buffer during the specified subpass. If `variableSampleLocations` is `VK_TRUE`, the sample locations used for rasterization do not depend on `pPostSubpassSampleLocations`.

### Valid Usage (Implicit)

- **VUID-VkRenderPassSampleLocationsBeginInfoEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT`

- **VUID-VkRenderPassSampleLocationsBeginInfoEXT-pAttachmentInitialSampleLocations-parameter**
  
  If `attachmentInitialSampleLocationsCount` is not 0, `pAttachmentInitialSampleLocations` must be a valid pointer to an array of `attachmentInitialSampleLocationsCount` valid `VkAttachmentSampleLocationsEXT` structures

- **VUID-VkRenderPassSampleLocationsBeginInfoEXT-pPostSubpassSampleLocations-parameter**
  
  If `postSubpassSampleLocationsCount` is not 0, `pPostSubpassSampleLocations` must be a valid pointer to an array of `postSubpassSampleLocationsCount` valid `VkSubpassSampleLocationsEXT` structures

The `VkAttachmentSampleLocationsEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkAttachmentSampleLocationsEXT {
    uint32_t attachmentIndex;
    VkSampleLocationsInfoEXT sampleLocationsInfo;
} VkAttachmentSampleLocationsEXT;
```

• **attachmentIndex** is the index of the attachment for which the sample locations state is provided.
• `sampleLocationsInfo` is the sample locations state to use for the layout transition of the given attachment from the initial layout of the attachment to the image layout specified for the attachment in the first subpass using it.

If the image referenced by the framebuffer attachment at index `attachmentIndex` was not created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` then the values specified in `sampleLocationsInfo` are ignored.

### Valid Usage

- `VUID-VkAttachmentSampleLocationsEXT-attachmentIndex-01531`  
  `attachmentIndex` must be less than the `attachmentCount` specified in `VkRenderPassCreateInfo` the render pass specified by `VkRenderPassBeginInfo::renderPass` was created with

### Valid Usage (Implicit)

- `VUID-VkAttachmentSampleLocationsEXT-sampleLocationsInfo-parameter`  
  `sampleLocationsInfo` must be a valid `VkSampleLocationsInfoEXT` structure

The `VkSubpassSampleLocationsEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkSubpassSampleLocationsEXT {
    uint32_t subpassIndex;
    VkSampleLocationsInfoEXT sampleLocationsInfo;
} VkSubpassSampleLocationsEXT;
```

- `subpassIndex` is the index of the subpass for which the sample locations state is provided.
- `sampleLocationsInfo` is the sample locations state to use for the layout transition of the depth/stencil attachment away from the image layout the attachment is used with in the subpass specified in `subpassIndex`.

If the image referenced by the depth/stencil attachment used in the subpass identified by `subpassIndex` was not created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` or if the subpass does not use a depth/stencil attachment, and `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_TRUE` then the values specified in `sampleLocationsInfo` are ignored.

### Valid Usage

- `VUID-VkSubpassSampleLocationsEXT-subpassIndex-01532`  
  `subpassIndex` must be less than the `subpassCount` specified in `VkRenderPassCreateInfo` the render pass specified by `VkRenderPassBeginInfo::renderPass` was created with
Valid Usage (Implicit)

- VUID-VkSubpassSampleLocationsEXT-sampleLocationsInfo-parameter
  sampleLocationsInfo must be a valid VkSampleLocationsInfoEXT structure

To begin a renderpass instance with render pass transform enabled, add the VkRenderPassTransformBeginInfoQCOM to the pNext chain of VkRenderPassBeginInfo structure passed to the vkCmdBeginRenderPass command specifying the renderpass transform.

The VkRenderPassTransformBeginInfoQCOM structure is defined as:

```c
// Provided by VK_QCOM_render_pass_transform
typedef struct VkRenderPassTransformBeginInfoQCOM {
    VkStructureType sType;
    void* pNext;
    VkSurfaceTransformFlagBitsKHR transform;
} VkRenderPassTransformBeginInfoQCOM;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- transform is a VkSurfaceTransformFlagBitsKHR value describing the transform to be applied to rasterization.

Valid Usage

- VUID-VkRenderPassTransformBeginInfoQCOM-transform-02871
  transform must be VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR, VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR, VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR, or VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR

- VUID-VkRenderPassTransformBeginInfoQCOM-flags-02872
  The renderpass must have been created with VkRenderPassCreateInfo::flags containing VK_RENDER_PASS_CREATE_TRANSFORM_BIT_QCOM

Valid Usage (Implicit)

- VUID-VkRenderPassTransformBeginInfoQCOM-sType-sType
  sType must be VK_STRUCTURE_TYPE_RENDER_PASS_TRANSFORM_BEGIN_INFO_QCOM

The VkSubpassBeginInfo structure is defined as:
```c
typedef struct VkSubpassBeginInfo {
    VkStructureType sType;
    const void* pNext;
    VkSubpassContents contents;
} VkSubpassBeginInfo;
```

or the equivalent

```c
// Provided by VK_KHR_create_renderpass2
typedef VkSubpassBeginInfo VkSubpassBeginInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `contents` is a `VkSubpassContents` value specifying how the commands in the next subpass will be provided.

**Valid Usage (Implicit)**

- `VUID-VkSubpassBeginInfo-sType-sType`  
  `sType` **must** be `VK_STRUCTURE_TYPE_SUBPASS_BEGIN_INFO`
- `VUID-VkSubpassBeginInfo-pNext-pNext`  
  `pNext` **must** be `NULL`
- `VUID-VkSubpassBeginInfo-contents-parameter`  
  `contents` **must** be a valid `VkSubpassContents` value

Possible values of `vkCmdBeginRenderPass::contents`, specifying how the commands in the first subpass will be provided, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSubpassContents {
    VK_SUBPASS_CONTENTS_INLINE = 0,
    VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS = 1,
} VkSubpassContents;
```

- `VK_SUBPASS_CONTENTS_INLINE` specifies that the contents of the subpass will be recorded inline in the primary command buffer, and secondary command buffers **must** not be executed within the subpass.
- `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS` specifies that the contents are recorded in secondary command buffers that will be called from the primary command buffer, and `vkCmdExecuteCommands` is the only valid command on the command buffer until `vkCmdNextSubpass` or `vkCmdEndRenderPass`.

If the `pNext` chain of `VkRenderPassBeginInfo` includes a `VkDeviceGroupRenderPassBeginInfo` structure,
then that structure includes a device mask and set of render areas for the render pass instance.

The `VkDeviceGroupRenderPassBeginInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDeviceGroupRenderPassBeginInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceMask;
    uint32_t           deviceRenderAreaCount;
    const VkRect2D*    pDeviceRenderAreas;
} VkDeviceGroupRenderPassBeginInfo;
```

or the equivalent

```c
// Provided by VK_KHR_device_group
typedef VkDeviceGroupRenderPassBeginInfo VkDeviceGroupRenderPassBeginInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceMask** is the device mask for the render pass instance.
- **deviceRenderAreaCount** is the number of elements in the `pDeviceRenderAreas` array.
- **pDeviceRenderAreas** is a pointer to an array of `VkRect2D` structures defining the render area for each physical device.

The **deviceMask** serves several purposes. It is an upper bound on the set of physical devices that can be used during the render pass instance, and the initial device mask when the render pass instance begins. In addition, commands transitioning to the next subpass in the render pass instance and commands ending the render pass instance, and, accordingly render pass attachment load, store, and resolve operations and subpass dependencies corresponding to the render pass instance, are executed on the physical devices included in the device mask provided here.

If **deviceRenderAreaCount** is not zero, then the elements of **pDeviceRenderAreas** override the value of `VkRenderPassBeginInfo::renderArea`, and provide a render area specific to each physical device. These render areas serve the same purpose as `VkRenderPassBeginInfo::renderArea`, including controlling the region of attachments that are cleared by `VK_ATTACHMENT_LOAD_OP_CLEAR` and that are resolved into resolve attachments.

If this structure is not present, the render pass instance's device mask is the value of `VkDeviceGroupCommandBufferBeginInfo::deviceMask`. If this structure is not present or if **deviceRenderAreaCount** is zero, `VkRenderPassBeginInfo::renderArea` is used for all physical devices.
Valid Usage

- VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00905
deviceMask must be a valid device mask value
- VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00906
deviceMask must not be zero
- VUID-VkDeviceGroupRenderPassBeginInfo-deviceMask-00907
deviceMask must be a subset of the command buffer’s initial device mask
- VUID-VkDeviceGroupRenderPassBeginInfo-deviceRenderAreaCount-00908
deviceRenderAreaCount must either be zero or equal to the number of physical devices in the logical device

Valid Usage (Implicit)

- VUID-VkDeviceGroupRenderPassBeginInfo-sType-sType
sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO
- VUID-VkDeviceGroupRenderPassBeginInfo-pDeviceRenderAreas-parameter
If deviceRenderAreaCount is not 0, pDeviceRenderAreas must be a valid pointer to an array of deviceRenderAreaCount VkRect2D structures

The VkRenderPassAttachmentBeginInfo structure is defined as:

```c
typedef struct VkRenderPassAttachmentBeginInfo {
    VkStructureType       sType;
    const void*           pNext;
    uint32_t              attachmentCount;
    const VkImageView*    pAttachments;
} VkRenderPassAttachmentBeginInfo;
```

or the equivalent

```c
// Provided by VK_KHR_imageless_framebuffer
typedef VkRenderPassAttachmentBeginInfo VkRenderPassAttachmentBeginInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- attachmentCount is the number of attachments.
- pAttachments is a pointer to an array of VkImageView handles, each of which will be used as the corresponding attachment in the render pass instance.
Valid Usage

- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-03218
  Each element of `pAttachments` must only specify a single mip level

- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-03219
  Each element of `pAttachments` must have been created with the identity swizzle

- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-04114
  Each element of `pAttachments` must have been created with `VkImageViewCreateInfo` `::viewType` not equal to `VK_IMAGE_VIEW_TYPE_3D`

Valid Usage (Implicit)

- VUID-VkRenderPassAttachmentBeginInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO`

- VUID-VkRenderPassAttachmentBeginInfo-pAttachments-parameter
  If `attachmentCount` is not 0, `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkImageView` handles

To query the render area granularity, call:

```c
// Provided by VK_VERSION_1_0
void vkGetRenderAreaGranularity(
    VkDevice device,
    VkRenderPass renderPass,
    VkExtent2D* pGranularity);
```

- `device` is the logical device that owns the render pass.
- `renderPass` is a handle to a render pass.
- `pGranularity` is a pointer to a `VkExtent2D` structure in which the granularity is returned.

The conditions leading to an optimal `renderArea` are:

- the `offset.x` member in `renderArea` is a multiple of the `width` member of the returned `VkExtent2D` (the horizontal granularity).
- the `offset.y` member in `renderArea` is a multiple of the `height` of the returned `VkExtent2D` (the vertical granularity).
- either the `offset.width` member in `renderArea` is a multiple of the horizontal granularity or `offset.x+offset.width` is equal to the `width` of the framebuffer in the `VkRenderPassBeginInfo`.
- either the `offset.height` member in `renderArea` is a multiple of the vertical granularity or `offset.y+offset.height` is equal to the `height` of the framebuffer in the `VkRenderPassBeginInfo`.

Subpass dependencies are not affected by the render area, and apply to the entire image
subresources attached to the framebuffer as specified in the description of automatic layout transitions. Similarly, pipeline barriers are valid even if their effect extends outside the render area.

Valid Usage (Implicit)

- `VUID-vkGetRenderAreaGranularity-device-parameter`
  - `device` must be a valid `VkDevice` handle

- `VUID-vkGetRenderAreaGranularity-renderPass-parameter`
  - `renderPass` must be a valid `VkRenderPass` handle

- `VUID-vkGetRenderAreaGranularity-pGranularity-parameter`
  - `pGranularity` must be a valid pointer to a `VkExtent2D` structure

- `VUID-vkGetRenderAreaGranularity-renderPass-parent`
  - `renderPass` must have been created, allocated, or retrieved from `device`

To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:

```c
// Provided by VK_VERSION_1_0
dataDefine void vkCmdNextSubpass(
    VkCommandBuffer commandBuffer,
    VkSubpassContents contents);
```

- `commandBuffer` is the command buffer in which to record the command.
- `contents` specifies how the commands in the next subpass will be provided, in the same fashion as the corresponding parameter of `vkCmdBeginRenderPass`.

The subpass index for a render pass begins at zero when `vkCmdBeginRenderPass` is recorded, and increments each time `vkCmdNextSubpass` is recorded.

Moving to the next subpass automatically performs any multisample resolve operations in the subpass being ended. End-of-subpass multisample resolves are treated as color attachment writes for the purposes of synchronization. This applies to resolve operations for both color and depth/stencil attachments. That is, they are considered to execute in the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` pipeline stage and their writes are synchronized with `VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT`. Synchronization between rendering within a subpass and any resolve operations at the end of the subpass occurs automatically, without need for explicit dependencies or pipeline barriers. However, if the resolve attachment is also used in a different subpass, an explicit dependency is needed.

After transitioning to the next subpass, the application can record the commands for that subpass.
Valid Usage

- VUID-vkCmdNextSubpass-None-00909
  The current subpass index must be less than the number of subpasses in the render pass minus one

- VUID-vkCmdNextSubpass-None-02349
  This command must not be recorded when transform feedback is active

Valid Usage (Implicit)

- VUID-vkCmdNextSubpass-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdNextSubpass-contents-parameter
  contents must be a valid VkSubpassContents value

- VUID-vkCmdNextSubpass-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdNextSubpass-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdNextSubpass-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdNextSubpass-bufferlevel
  commandBuffer must be a primary VkCommandBuffer

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

To transition to the next subpass in the render pass instance after recording the commands for a subpass, call:
void vkCmdNextSubpass2(
    VkCommandBuffer commandBuffer,
    const VkSubpassBeginInfo* pSubpassBeginInfo,
    const VkSubpassEndInfo* pSubpassEndInfo);

// Provided by VK_KHR_create_renderpass2
void vkCmdNextSubpass2KHR(
    VkCommandBuffer commandBuffer,
    const VkSubpassBeginInfo* pSubpassBeginInfo,
    const VkSubpassEndInfo* pSubpassEndInfo);

- `commandBuffer` is the command buffer in which to record the command.
- `pSubpassBeginInfo` is a pointer to a `VkSubpassBeginInfo` structure containing information about the subpass which is about to begin rendering.
- `pSubpassEndInfo` is a pointer to a `VkSubpassEndInfo` structure containing information about how the previous subpass will be ended.

`vkCmdNextSubpass2` is semantically identical to `vkCmdNextSubpass`, except that it is extensible, and that `contents` is provided as part of an extensible structure instead of as a flat parameter.

## Valid Usage

- VUID-vkCmdNextSubpass2-None-03102
  The current subpass index **must** be less than the number of subpasses in the render pass minus one

- VUID-vkCmdNextSubpass2-None-02350
  This command **must** not be recorded when transform feedback is active
Valid Usage (Implicit)

- VUID-vkCmdNextSubpass2-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdNextSubpass2-pSubpassBeginInfo-parameter
  `pSubpassBeginInfo` must be a valid pointer to a valid `VkSubpassBeginInfo` structure

- VUID-vkCmdNextSubpass2-pSubpassEndInfo-parameter
  `pSubpassEndInfo` must be a valid pointer to a valid `VkSubpassEndInfo` structure

- VUID-vkCmdNextSubpass2-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdNextSubpass2-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdNextSubpass2-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdNextSubpass2-bufferlevel
  `commandBuffer` must be a primary `VkCommandBuffer`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

To record a command to end a render pass instance after recording the commands for the last subpass, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdEndRenderPass(
    VkCommandBuffer commandBuffer);
```

- `commandBuffer` is the command buffer in which to end the current render pass instance.

Ending a render pass instance performs any multisample resolve operations on the final subpass.
Valid Usage

- VUID-vkCmdEndRenderPass-None-00910
  The current subpass index must be equal to the number of subpasses in the render pass minus one

- VUID-vkCmdEndRenderPass-None-02351
  This command must not be recorded when transform feedback is active

Valid Usage (Implicit)

- VUID-vkCmdEndRenderPass-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdEndRenderPass-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdEndRenderPass-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdEndRenderPass-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdEndRenderPass-bufferlevel
  commandBuffer must be a primary VkCommandBuffer

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

To record a command to end a render pass instance after recording the commands for the last subpass, call:
void vkCmdEndRenderPass2(
    VkCommandBuffer commandBuffer,
    const VkSubpassEndInfo* pSubpassEndInfo);

// Provided by VK_KHR_create_renderpass2
void vkCmdEndRenderPass2KHR(
    VkCommandBuffer commandBuffer,
    const VkSubpassEndInfo* pSubpassEndInfo);

- `commandBuffer` is the command buffer in which to end the current render pass instance.
- `pSubpassEndInfo` is a pointer to a `VkSubpassEndInfo` structure containing information about how the previous subpass will be ended.

`vkCmdEndRenderPass2` is semantically identical to `vkCmdEndRenderPass`, except that it is extensible.

### Valid Usage

- **VUID-vkCmdEndRenderPass2-None-03103**
  The current subpass index must be equal to the number of subpasses in the render pass minus one
- **VUID-vkCmdEndRenderPass2-None-02352**
  This command must not be recorded when transform feedback is active

### Valid Usage (Implicit)

- **VUID-vkCmdEndRenderPass2-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- **VUID-vkCmdEndRenderPass2-pSubpassEndInfo-parameter**
  `pSubpassEndInfo` must be a valid pointer to a valid `VkSubpassEndInfo` structure
- **VUID-vkCmdEndRenderPass2-commandBuffer-recording**
  `commandBuffer` must be in the recording state
- **VUID-vkCmdEndRenderPass2-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- **VUID-vkCmdEndRenderPass2-renderpass**
  This command must only be called inside of a render pass instance
- **VUID-vkCmdEndRenderPass2-bufferlevel**
  `commandBuffer` must be a primary `VkCommandBuffer`
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

The `VkSubpassEndInfo` structure is defined as:

```c
typedef struct VkSubpassEndInfo {
    VkStructureType    sType;
    const void*        pNext;
} VkSubpassEndInfo;
```

or the equivalent

```
// Provided by VK_KHR_create_renderpass2
typedef VkSubpassEndInfo VkSubpassEndInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.

Valid Usage (Implicit)

- VUID-VkSubpassEndInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SUBPASS_END_INFO`
- VUID-VkSubpassEndInfo-pNext-pNext
  `pNext` must be `NULL`
Chapter 9. Shaders

A shader specifies programmable operations that execute for each vertex, control point, tessellated vertex, primitive, fragment, or workgroup in the corresponding stage(s) of the graphics and compute pipelines.

Graphics pipelines include vertex shader execution as a result of primitive assembly, followed, if enabled, by tessellation control and evaluation shaders operating on patches, geometry shaders, if enabled, operating on primitives, and fragment shaders, if present, operating on fragments generated by Rasterization. In this specification, vertex, tessellation control, tessellation evaluation and geometry shaders are collectively referred to as vertex processing stages and occur in the logical pipeline before rasterization. The fragment shader occurs logically after rasterization.

Only the compute shader stage is included in a compute pipeline. Compute shaders operate on compute invocations in a workgroup.

Shaders can read from input variables, and read from and write to output variables. Input and output variables can be used to transfer data between shader stages, or to allow the shader to interact with values that exist in the execution environment. Similarly, the execution environment provides constants that describe capabilities.

Shader variables are associated with execution environment-provided inputs and outputs using built-in decorations in the shader. The available decorations for each stage are documented in the following subsections.

9.1. Shader Modules

Shader modules contain shader code and one or more entry points. Shaders are selected from a shader module by specifying an entry point as part of pipeline creation. The stages of a pipeline can use shaders that come from different modules. The shader code defining a shader module must be in the SPIR-V format, as described by the Vulkan Environment for SPIR-V appendix.

Shader modules are represented by VkShaderModule handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkShaderModule)
```

To create a shader module, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateShaderModule(
    VkDevice device,
    const VkShaderModuleCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkShaderModule* pShaderModule);
```

- `device` is the logical device that creates the shader module.
• **pCreateInfo** is a pointer to a `VkShaderModuleCreateInfo` structure.
• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
• **pShaderModule** is a pointer to a `VkShaderModule` handle in which the resulting shader module object is returned.

Once a shader module has been created, any entry points it contains can be used in pipeline shader stages as described in Compute Pipelines and Graphics Pipelines.

If the shader stage fails to compile `VK_ERROR_INVALID_SHADER_NV` will be generated and the compile log will be reported back to the application by `VK_EXT_debug_report` if enabled.

### Valid Usage (Implicit)

- **VUID-vkCreateShaderModule-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkCreateShaderModule-pCreateInfo-parameter**
  - `pCreateInfo` must be a valid pointer to a valid `VkShaderModuleCreateInfo` structure
- **VUID-vkCreateShaderModule-pAllocator-parameter**
  - If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateShaderModule-pShaderModule-parameter**
  - `pShaderModule` must be a valid pointer to a `VkShaderModule` handle

### Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OFDEVICE_MEMORY**
- **VK_ERROR_INVALID_SHADER_NV**

The `VkShaderModuleCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkShaderModuleCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkShaderModuleCreateFlags flags;
    size_t                    codeSize;
    const uint32_t*           pCode;
} VkShaderModuleCreateInfo;
```
- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `codeSize` is the size, in bytes, of the code pointed to by `pCode`.
- `pCode` is a pointer to code that is used to create the shader module. The type and format of the code is determined from the content of the memory addressed by `pCode`.

**Valid Usage**

- VUID-VkShaderModuleCreateInfo-codeSize-01085
  
  `codeSize` **must** be greater than 0

- VUID-VkShaderModuleCreateInfo-pCode-01376
  
  If `pCode` is a pointer to SPIR-V code, `codeSize` **must** be a multiple of 4

- VUID-VkShaderModuleCreateInfo-pCode-01377
  
  `pCode` **must** point to either valid SPIR-V code, formatted and packed as described by the Khronos SPIR-V Specification or valid GLSL code which **must** be written to the GL_KHR_vulkan_gls1 extension specification

- VUID-VkShaderModuleCreateInfo-pCode-01378
  
  If `pCode` is a pointer to SPIR-V code, that code **must** adhere to the validation rules described by the Validation Rules within a Module section of the SPIR-V Environment appendix

- VUID-VkShaderModuleCreateInfo-pCode-01379
  
  If `pCode` is a pointer to GLSL code, it **must** be valid GLSL code written to the GL_KHR_vulkan_gls1 GLSL extension specification

- VUID-VkShaderModuleCreateInfo-pCode-01089
  
  `pCode` **must** declare the Shader capability for SPIR-V code

- VUID-VkShaderModuleCreateInfo-pCode-01090
  
  `pCode` **must** not declare any capability that is not supported by the API, as described by the Capabilities section of the SPIR-V Environment appendix

- VUID-VkShaderModuleCreateInfo-pCode-01091
  
  If `pCode` declares any of the capabilities listed in the SPIR-V Environment appendix, one of the corresponding requirements **must** be satisfied

- VUID-VkShaderModuleCreateInfo-pCode-04146
  
  `pCode` **must** not declare any SPIR-V extension that is not supported by the API, as described by the Extension section of the SPIR-V Environment appendix

- VUID-VkShaderModuleCreateInfo-pCode-04147
  
  If `pCode` declares any of the SPIR-V extensions listed in the SPIR-V Environment appendix, one of the corresponding requirements **must** be satisfied
Valid Usage (Implicit)

- **VUID-VkShaderModuleCreateInfo-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO`

- **VUID-VkShaderModuleCreateInfo-pNext-pNext**
  
  *pNext* must be `NULL` or a pointer to a valid instance of `VkShaderModuleValidationCacheCreateInfoEXT`

- **VUID-VkShaderModuleCreateInfo-sType-unique**
  
  The *sType* value of each struct in the *pNext* chain must be unique

- **VUID-VkShaderModuleCreateInfo-flags-zerobitmask**
  
  *flags* must be `0`

- **VUID-VkShaderModuleCreateInfo-pCode-parameter**
  
  *pCode* must be a valid pointer to an array of \( \frac{codeSize}{4} \) `uint32_t` values

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkShaderModuleCreateFlags;
```

`VkShaderModuleCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To use a `VkValidationCacheEXT` to cache shader validation results, add a `VkShaderModuleValidationCacheCreateInfoEXT` structure to the `pNext` chain of the `VkShaderModuleCreateInfo` structure, specifying the cache object to use.

The `VkShaderModuleValidationCacheCreateInfoEXT` struct is defined as:

```c
// Provided by VK_EXT_validation_cache
typedef struct VkShaderModuleValidationCacheCreateInfoEXT {
    VkStructureType          sType;
    const void*              pNext;
    VkValidationCacheEXT     validationCache;
} VkShaderModuleValidationCacheCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **validationCache** is the validation cache object from which the results of prior validation attempts will be written, and to which new validation results for this `VkShaderModule` will be written (if not already present).
Valid Usage (Implicit)

- **VUID-VkShaderModuleValidationCacheCreateInfoEXT-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_SHADER_MODULE_VALIDATION_CACHE_CREATE_INFO_EXT`

- **VUID-VkShaderModuleValidationCacheCreateInfoEXT-validationCache-parameter**
  - `validationCache` **must** be a valid `VkValidationCacheEXT` handle

To destroy a shader module, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyShaderModule(
    VkDevice                                    device,
    VkShaderModule                              shaderModule,
    const VkAllocationCallbacks*                pAllocator);
```

- `device` is the logical device that destroys the shader module.
- `shaderModule` is the handle of the shader module to destroy.
- `pAllocator` controls host memory allocation as described in the [Memory Allocation](#) chapter.

A shader module **can** be destroyed while pipelines created using its shaders are still in use.

Valid Usage

- **VUID-vkDestroyShaderModule-shaderModule-01092**
  - If `VkAllocationCallbacks` were provided when `shaderModule` was created, a compatible set of callbacks **must** be provided here

- **VUID-vkDestroyShaderModule-shaderModule-01093**
  - If no `VkAllocationCallbacks` were provided when `shaderModule` was created, `pAllocator` **must** be `NULL`
Valid Usage (Implicit)

- VUID-vkDestroyShaderModule-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyShaderModule-shaderModule-parameter
  If shaderModule is not VK_NULL_HANDLE, shaderModule must be a valid VkShaderModule handle

- VUID-vkDestroyShaderModule-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyShaderModule-shaderModule-parent
  If shaderModule is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to shaderModule must be externally synchronized

9.2. Shader Execution

At each stage of the pipeline, multiple invocations of a shader may execute simultaneously. Further, invocations of a single shader produced as the result of different commands may execute simultaneously. The relative execution order of invocations of the same shader type is undefined. Shader invocations may complete in a different order than that in which the primitives they originated from were drawn or dispatched by the application. However, fragment shader outputs are written to attachments in rasterization order.

The relative execution order of invocations of different shader types is largely undefined. However, when invoking a shader whose inputs are generated from a previous pipeline stage, the shader invocations from the previous stage are guaranteed to have executed far enough to generate input values for all required inputs.

9.3. Shader Memory Access Ordering

The order in which image or buffer memory is read or written by shaders is largely undefined. For some shader types (vertex, tessellation evaluation, and in some cases, fragment), even the number of shader invocations that may perform loads and stores is undefined.

In particular, the following rules apply:

- Vertex and tessellation evaluation shaders will be invoked at least once for each unique vertex, as defined in those sections.
- Fragment shaders will be invoked zero or more times, as defined in that section.
- The relative execution order of invocations of the same shader type is undefined. A store issued
by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are always written to the framebuffer in rasterization order, stores executed by fragment shader invocations are not.

- The relative execution order of invocations of different shader types is largely undefined.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The above limitations on shader invocation order make some forms of synchronization between shader invocations within a single set of primitives unimplementable. For example, having one invocation poll memory written by another invocation assumes that the other invocation has been launched and will complete its writes in finite time.</td>
</tr>
</tbody>
</table>

The Memory Model appendix defines the terminology and rules for how to correctly communicate between shader invocations, such as when a write is Visible-To a read, and what constitutes a Data Race.

Applications must not cause a data race.

The SPIR-V SubgroupMemory, CrossWorkgroupMemory, and AtomicCounterMemory memory semantics are ignored. Sequentially consistent atomics and barriers are not supported and SequentiallyConsistent is treated as AcquireRelease. SequentiallyConsistent should not be used.

9.4. Shader Inputs and Outputs

Data is passed into and out of shaders using variables with input or output storage class, respectively. User-defined inputs and outputs are connected between stages by matching their Location decorations. Additionally, data can be provided by or communicated to special functions provided by the execution environment using BuiltIn decorations.

In many cases, the same BuiltIn decoration can be used in multiple shader stages with similar meaning. The specific behavior of variables decorated as BuiltIn is documented in the following sections.

9.5. Task Shaders

Task shaders operate in conjunction with the mesh shaders to produce a collection of primitives that will be processed by subsequent stages of the graphics pipeline. Its primary purpose is to create a variable amount of subsequent mesh shader invocations.

Task shaders are invoked via the execution of the programmable mesh shading pipeline.

The task shader has no fixed-function inputs other than variables identifying the specific workgroup and invocation. The only fixed output of the task shader is a task count, identifying the number of mesh shader workgroups to create. The task shader can write additional outputs to task memory, which can be read by all of the mesh shader workgroups it created.
9.5.1. Task Shader Execution

Task workloads are formed from groups of work items called workgroups and processed by the task shader in the current graphics pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Task shaders execute in global workgroups which are divided into a number of local workgroups with a size that can be set by assigning a value to the LocalSize execution mode or via an object decorated by the WorkgroupSize decoration. An invocation within a local workgroup can share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

9.6. Mesh Shaders

Mesh shaders operate in workgroups to produce a collection of primitives that will be processed by subsequent stages of the graphics pipeline. Each workgroup emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.

Mesh shaders are invoked via the execution of the programmable mesh shading pipeline.

The only inputs available to the mesh shader are variables identifying the specific workgroup and invocation and, if applicable, any outputs written to task memory by the task shader that spawned the mesh shader’s workgroup. The mesh shader can operate without a task shader as well.

The invocations of the mesh shader workgroup write an output mesh, comprising a set of primitives with per-primitive attributes, a set of vertices with per-vertex attributes, and an array of indices identifying the mesh vertices that belong to each primitive. The primitives of this mesh are then processed by subsequent graphics pipeline stages, where the outputs of the mesh shader form an interface with the fragment shader.

9.6.1. Mesh Shader Execution

Mesh workloads are formed from groups of work items called workgroups and processed by the mesh shader in the current graphics pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Mesh shaders execute in global workgroups which are divided into a number of local workgroups with a size that can be set by assigning a value to the LocalSize execution mode or via an object decorated by the WorkgroupSize decoration. An invocation within a local workgroup can share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

The global workgroups may be generated explicitly via the API, or implicitly through the task shader’s work creation mechanism.

9.7. Vertex Shaders

Each vertex shader invocation operates on one vertex and its associated vertex attribute data, and outputs one vertex and associated data. Graphics pipelines using primitive shading must include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.
9.7.1. Vertex Shader Execution

A vertex shader must be executed at least once for each vertex specified by a draw command. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view. During execution, the shader is presented with the index of the vertex and instance for which it has been invoked. Input variables declared in the vertex shader are filled by the implementation with the values of vertex attributes associated with the invocation being executed.

If the same vertex is specified multiple times in a draw command (e.g. by including the same index value multiple times in an index buffer) the implementation may reuse the results of vertex shading if it can statically determine that the vertex shader invocations will produce identical results.

Note

It is implementation-dependent when and if results of vertex shading are reused, and thus how many times the vertex shader will be executed. This is true also if the vertex shader contains stores or atomic operations (see vertexPipelineStoresAndAtomics).

9.8. Tessellation Control Shaders

The tessellation control shader is used to read an input patch provided by the application and to produce an output patch. Each tessellation control shader invocation operates on an input patch (after all control points in the patch are processed by a vertex shader) and its associated data, and outputs a single control point of the output patch and its associated data, and can also output additional per-patch data. The input patch is sized according to the patchControlPoints member of VkPipelineTessellationStateCreateInfo, as part of input assembly.

The input patch can also be dynamically sized with patchControlPoints parameter of vkCmdSetPatchControlPointsEXT.

To dynamically specify the number of control points per patch:

```c
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetPatchControlPointsEXT(
    VkCommandBuffer commandBuffer, 
    uint32_t patchControlPoints);
```

- commandBuffer is the command buffer into which the command will be recorded.
- patchControlPoints specifies number of control points per patch.

This command sets the state for a given draw when the graphics pipeline is created with VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT set in VkPipelineDynamicStateCreateInfo::pDynamicStates.
Valid Usage

- VUID-vkCmdSetPatchControlPointsEXT-None-04873
  The extendedDynamicState2PatchControlPoints feature must be enabled

- VUID-vkCmdSetPatchControlPointsEXT-patchControlPoints-04874
  patchControlPoints must be greater than zero and less than or equal to VkPhysicalDeviceLimits::maxTessellationPatchSize

Valid Usage (Implicit)

- VUID-vkCmdSetPatchControlPointsEXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdSetPatchControlPointsEXT-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdSetPatchControlPointsEXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The size of the output patch is controlled by the OpExecutionMode OutputVertices specified in the tessellation control or tessellation evaluation shaders, which must be specified in at least one of the shaders. The size of the input and output patches must each be greater than zero and less than or equal to VkPhysicalDeviceLimits::maxTessellationPatchSize.

9.8.1. Tessellation Control Shader Execution

A tessellation control shader is invoked at least once for each output vertex in a patch. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.
Inputs to the tessellation control shader are generated by the vertex shader. Each invocation of the tessellation control shader can read the attributes of any incoming vertices and their associated data. The invocations corresponding to a given patch execute logically in parallel, with undefined relative execution order. However, the `OpControlBarrier` instruction can be used to provide limited control of the execution order by synchronizing invocations within a patch, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will read undefined values if one invocation reads a per-vertex or per-patch attribute written by another invocation at any point during the same phase, or if two invocations attempt to write different values to the same per-patch output in a single phase.

### 9.9. Tessellation Evaluation Shaders

The Tessellation Evaluation Shader operates on an input patch of control points and their associated data, and a single input barycentric coordinate indicating the invocation's relative position within the subdivided patch, and outputs a single vertex and its associated data.

#### 9.9.1. Tessellation Evaluation Shader Execution

A tessellation evaluation shader is invoked at least once for each unique vertex generated by the tessellator. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

### 9.10. Geometry Shaders

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.

#### 9.10.1. Geometry Shader Execution

A geometry shader is invoked at least once for each primitive produced by the tessellation stages, or at least once for each primitive generated by `primitive assembly` when tessellation is not in use. A shader can request that the geometry shader runs multiple instances. A geometry shader is invoked at least once for each instance. If the subpass includes multiple views in its view mask, the shader may be invoked separately for each view.

### 9.11. Fragment Shaders

Fragment shaders are invoked as the result of rasterization in a graphics pipeline. Each fragment shader invocation operates on a single fragment and its associated data. With few exceptions, fragment shaders do not have access to any data associated with other fragments and are considered to execute in isolation of fragment shader invocations associated with other fragments.

#### 9.11.1. Fragment Shader Execution

Fragment shaders are invoked for each fragment generated by rasterization, or as helper invocations.
For fragment shaders invoked by fragments, the following rules apply:

- A fragment shader **must** not be executed if a fragment operation that executes before fragment shading discards the fragment.

- A fragment shader **may** not be executed if:
  
  ◦ An implementation determines that another fragment shader, invoked by a subsequent primitive in primitive order, overwrites all results computed by the shader (including writes to storage resources).
  
  ◦ Any other fragment operation discards the fragment, and the shader does not write to any storage resources.

- Otherwise, at least one fragment shader **must** be executed.
  
  ◦ If sample shading is enabled and multiple invocations per fragment are **required**, additional invocations **must** be executed as specified.
  
  ◦ If a shading rate image is used and multiple invocations per fragment are **required**, additional invocations **must** be executed as specified.
  
  ◦ Each covered sample **must** be included in at least one fragment shader invocation.

---

**Note**

Multiple fragment shader invocations may be executed for the same fragment for any number of implementation dependent reasons. When there is more than one fragment shader invocation per fragment, the association of samples to invocations is implementation-dependent. Stores and atomics performed by these additional invocations have the normal effect.

For example, if the subpass includes multiple views in its view mask, a fragment shader may be invoked separately for each view.

Similarly, if the render pass has a fragment density map attachment, more than one fragment shader invocation may be invoked for each covered sample. Such additional invocations are only produced if 

```
VkPhysicalDeviceFragmentDensityMapPropertiesEXT::fragmentDensityInvocations
```

is **VK_TRUE**. Implementations may generate these additional fragment shader invocations in order to make transitions between fragment areas with different fragment densities more smooth.

---

**Note**

Relative ordering of execution of different fragment shader invocations is explicitly not defined.

### 9.11.2. Early Fragment Tests

An explicit control is provided to allow fragment shaders to enable early fragment tests. If the fragment shader specifies the `EarlyFragmentTests OpExecutionMode`, additional per-fragment tests are performed prior to fragment shader execution.
If the fragment shader additionally specifies the `PostDepthCoverage OpExecutionMode`, the value of a variable decorated with the `SampleMask` built-in reflects the coverage after the early fragment tests. Otherwise, it reflects the coverage before the early fragment tests.

If early fragment tests are enabled, any depth value computed by the fragment shader has no effect.

### 9.11.3. Fragment Shader Interlock

In normal operation, it is possible for more than one fragment shader invocation to be executed simultaneously for the same pixel if there are overlapping primitives. If the `fragmentShaderSampleInterlock`, `fragmentShaderPixelInterlock`, or `fragmentShaderShadingRateInterlock` features are enabled, it is possible to define a critical section within the fragment shader that is guaranteed to not run simultaneously with another fragment shader invocation for the same sample(s) or pixel(s). It is also possible to control the relative ordering of execution of these critical sections across different fragment shader invocations.

If the `FragmentShaderSampleInterlockEXT`, `FragmentShaderPixelInterlockEXT`, or `FragmentShaderShadingRateInterlockEXT` capabilities are declared in the fragment shader, the `OpBeginInvocationInterlockEXT` and `OpEndInvocationInterlockEXT` instructions **must** be used to delimit a critical section of fragment shader code.

To ensure each invocation of the critical section is executed in *primitive order*, declare one of the `PixelInterlockOrderedEXT`, `SampleInterlockOrderedEXT`, or `ShadingRateInterlockOrderedEXT` execution modes. If the order of execution of each invocation of the critical section does not matter, declare one of the `PixelInterlockUnorderedEXT`, `SampleInterlockUnorderedEXT`, or `ShadingRateInterlockUnorderedEXT` execution modes.

The `PixelInterlockOrderedEXT` and `PixelInterlockUnorderedEXT` execution modes provide mutual exclusion in the critical section for any pair of fragments corresponding to the same pixel, or pixels if the fragment covers more than one pixel. With sample shading enabled, these execution modes are treated like `SampleInterlockOrderedEXT` or `SampleInterlockUnorderedEXT` respectively.

The `SampleInterlockOrderedEXT` and `SampleInterlockUnorderedEXT` execution modes only provide mutual exclusion for pairs of fragments that both cover at least one common sample in the same pixel; these are recommended for performance if shaders use per-sample data structures. If these execution modes are used in single-sample mode they are treated like `PixelInterlockOrderedEXT` or `PixelInterlockUnorderedEXT` respectively.

The `ShadingRateInterlockOrderedEXT` and `ShadingRateInterlockUnorderedEXT` execution modes provide mutual exclusion for pairs of fragments that both have at least one common sample in the same pixel, even if none of the common samples are covered by both fragments. With sample shading enabled, these execution modes are treated like `SampleInterlockOrderedEXT` or `SampleInterlockUnorderedEXT` respectively.

### 9.12. Compute Shaders

Compute shaders are invoked via `vkCmdDispatch` and `vkCmdDispatchIndirect` commands. In general, they have access to similar resources as shader stages executing as part of a graphics pipeline.
Compute workloads are formed from groups of work items called workgroups and processed by the compute shader in the current compute pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Compute shaders execute in global workgroups which are divided into a number of local workgroups with a size that can be set by assigning a value to the LocalSize execution mode or via an object decorated by the WorkgroupSize decoration. An invocation within a local workgroup can share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

9.13. Ray Generation Shaders

A ray generation shader is similar to a compute shader. Its main purpose is to execute ray tracing queries using OpTraceRayKHR instructions and process the results.

9.13.1. Ray Generation Shader Execution

One ray generation shader is executed per ray tracing dispatch. Its location in the shader binding table (see Shader Binding Table for details) is passed directly into vkCmdTraceRaysKHR using the raygenShaderBindingTableBuffer and raygenShaderBindingOffset parameters.


Intersection shaders enable the implementation of arbitrary, application defined geometric primitives. An intersection shader for a primitive is executed whenever its axis-aligned bounding box is hit by a ray.

Like other ray tracing shader domains, an intersection shader operates on a single ray at a time. It also operates on a single primitive at a time. It is therefore the purpose of an intersection shader to compute the ray-primitive intersections and report them. To report an intersection, the shader calls the OpReportIntersectionKHR instruction.

An intersection shader communicates with any-hit and closest shaders by generating attribute values that they can read. Intersection shaders cannot read or modify the ray payload.

9.14.1. Intersection Shader Execution

The order in which intersections are found along a ray, and therefore the order in which intersection shaders are executed, is unspecified.

The intersection shader of the closest AABB which intersects the ray is guaranteed to be executed at some point during traversal, unless the ray is forcibly terminated.

9.15. Any-Hit Shaders

The any-hit shader is executed after the intersection shader reports an intersection that lies within the current \([t_{\text{min}}, t_{\text{max}}]\) of the ray. The main use of any-hit shaders is to programmatically decide whether or not an intersection will be accepted. The intersection will be accepted unless the shader calls the OpIgnoreIntersectionKHR instruction. Any-hit shaders have read-only access to the
attributes generated by the corresponding intersection shader, and can read or modify the ray payload.

9.15.1. Any-Hit Shader Execution

The order in which intersections are found along a ray, and therefore the order in which any-hit shaders are executed, is unspecified.

The any-hit shader of the closest hit is guaranteed to be executed at some point during traversal, unless the ray is forcibly terminated.

9.16. Closest Hit Shaders

Closest hit shaders have read-only access to the attributes generated by the corresponding intersection shader, and can read or modify the ray payload. They also have access to a number of system-generated values. Closest hit shaders can call OpTraceRayKHR to recursively trace rays.

9.16.1. Closest Hit Shader Execution

Exactly one closest hit shader is executed when traversal is finished and an intersection has been found and accepted.

9.17. Miss Shaders

Miss shaders can access the ray payload and can trace new rays through the OpTraceRayKHR instruction, but cannot access attributes since they are not associated with an intersection.

9.17.1. Miss Shader Execution

A miss shader is executed instead of a closest hit shader if no intersection was found during traversal.

9.18. Callable Shaders

Callable shaders can access a callable payload that works similarly to ray payloads to do subroutine work.

9.18.1. Callable Shader Execution

A callable shader is executed by calling OpExecuteCallableKHR from an allowed shader stage.

9.19. Interpolation Decorations

Interpolation decorations control the behavior of attribute interpolation in the fragment shader stage. Interpolation decorations can be applied to Input storage class variables in the fragment shader stage’s interface, and control the interpolation behavior of those variables.
Inputs that could be interpolated can be decorated by at most one of the following decorations:

- **Flat**: no interpolation
- **NoPerspective**: linear interpolation (for lines and polygons)

Fragment input variables decorated with neither Flat nor NoPerspective use perspective-correct interpolation (for lines and polygons).

The presence of and type of interpolation is controlled by the above interpolation decorations as well as the auxiliary decorations **Centroid** and **Sample**.

A variable decorated with Flat will not be interpolated. Instead, it will have the same value for every fragment within a triangle. This value will come from a single **provoking vertex**. A variable decorated with Flat can also be decorated with **Centroid** or **Sample**, which will mean the same thing as decorating it only as Flat.

For fragment shader input variables decorated with neither **Centroid** nor **Sample**, the assigned variable may be interpolated anywhere within the fragment and a single value may be assigned to each sample within the fragment.

If a fragment shader input is decorated with **Centroid**, a single value may be assigned to that variable for all samples in the fragment, but that value must be interpolated to a location that lies in both the fragment and in the primitive being rendered, including any of the fragment's samples covered by the primitive. Because the location at which the variable is interpolated may be different in neighboring fragments, and derivatives may be computed by computing differences between neighboring fragments, derivatives of centroid-sampled inputs may be less accurate than those for non-centroid interpolated variables. The **PostDepthCoverage** execution mode does not affect the determination of the centroid location.

If a fragment shader input is decorated with **Sample**, a separate value must be assigned to that variable for each covered sample in the fragment, and that value must be sampled at the location of the individual sample. When **rasterizationSamples** is **VK_SAMPLE_COUNT_1_BIT**, the fragment center must be used for **Centroid**, **Sample**, and undecorated attribute interpolation.

Fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type must be decorated with Flat.

When the **VK_AMD_shader_explicit_vertex_parameter** device extension is enabled inputs can be also decorated with the **CustomInterpAMD** interpolation decoration, including fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type. Inputs decorated with **CustomInterpAMD** can only be accessed by the extended instruction **InterpolateAtVertexAMD** and allows accessing the value of the input for individual vertices of the primitive.

When the **fragmentShaderBarycentric** feature is enabled, inputs can be also decorated with the **PerVertexNV** interpolation decoration, including fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type. Inputs decorated with **PerVertexNV** can only be accessed using an extra array dimension, where the extra index identifies one of the vertices of the primitive that produced the fragment.
9.20. Static Use

A SPIR-V module declares a global object in memory using the `OpVariable` instruction, which results in a pointer `x` to that object. A specific entry point in a SPIR-V module is said to `statically use` that object if that entry point's call tree contains a function containing a memory instruction or image instruction with `x` as an `id` operand. See the “Memory Instructions” and “Image Instructions” subsections of section 3 “Binary Form” of the SPIR-V specification for the complete list of SPIR-V memory instructions.

Static use is not used to control the behavior of variables with `Input` and `Output` storage. The effects of those variables are applied based only on whether they are present in a shader entry point’s interface.

9.21. Scope

A `scope` describes a set of shader invocations, where each such set is a `scope instance`. Each invocation belongs to one or more scope instances, but belongs to no more than one scope instance for each scope.

The operations available between invocations in a given scope instance vary, with smaller scopes generally able to perform more operations, and with greater efficiency.

9.21.1. Cross Device

All invocations executed in a Vulkan instance fall into a single `cross device scope instance`.

Whilst the `CrossDevice` scope is defined in SPIR-V, it is disallowed in Vulkan. API synchronization commands can be used to communicate between devices.

9.21.2. Device

All invocations executed on a single device form a `device scope instance`.

If the `vulkanMemoryModel` and `vulkanMemoryModelDeviceScope` features are enabled, this scope is represented in SPIR-V by the `Device Scope`, which can be used as a `Memory Scope` for barrier and atomic operations.

If both the `shaderDeviceClock` and `vulkanMemoryModelDeviceScope` features are enabled, using the `Device Scope` with the `OpReadClockKHR` instruction will read from a clock that is consistent across invocations in the same device scope instance.

There is no method to synchronize the execution of these invocations within SPIR-V, and this can only be done with API synchronization primitives.

Invocations executing on different devices in a device group operate in separate device scope instances.
9.21.3. Queue Family

Invocations executed by queues in a given queue family form a *queue family scope instance*.

This scope is identified in SPIR-V as the *QueueFamily Scope* if the *vulkanMemoryModel* feature is enabled, or if not, the *Device Scope*, which can be used as a *Memory Scope* for barrier and atomic operations.

If the *shaderDeviceClock* feature is enabled, but the *vulkanMemoryModelDeviceScope* feature is not enabled, using the *Device Scope* with the *OpReadClockKHR* instruction will read from a clock that is consistent across invocations in the same queue family scope instance.

There is no method to synchronize the execution of these invocations within SPIR-V, and this can only be done with API synchronization primitives.

Each invocation in a queue family scope instance must be in the same *device scope instance*.

9.21.4. Command

Any shader invocations executed as the result of a single command such as *vkCmdDispatch* or *vkCmdDraw* form a *command scope instance*. For indirect drawing commands with *drawCount* greater than one, invocations from separate draws are in separate command scope instances. For ray tracing shaders, an invocation group is an implementation-dependent subset of the set of shader invocations of a given shader stage which are produced by a single trace rays command.

There is no specific *Scope* for communication across invocations in a command scope instance. As this has a clear boundary at the API level, coordination here can be performed in the API, rather than in SPIR-V.

Each invocation in a command scope instance must be in the same *queue-family scope instance*.

For shaders without defined *workgroups*, this set of invocations forms an *invocation group* as defined in the SPIR-V specification.

9.21.5. Primitive

Any fragment shader invocations executed as the result of rasterization of a single primitive form a *primitive scope instance*.

There is no specific *Scope* for communication across invocations in a primitive scope instance.

Any generated *helper invocations* are included in this scope instance.

Each invocation in a primitive scope instance must be in the same *command scope instance*.

Any input variables decorated with *Flat* are uniform within a primitive scope instance.

9.21.6. Shader Call

Any *shader-call-related* invocations that are executed in one or more ray tracing execution models form a *shader call scope instance*. 
The ShaderCallKHR Scope can be used as Memory Scope for barrier and atomic operations.

Each invocation in a shader call scope instance must be in the same queue family scope instance.

### 9.21.7. Workgroup

A local workgroup is a set of invocations that can synchronize and share data with each other using memory in the Workgroup storage class.

The Workgroup Scope can be used as both an Execution Scope and Memory Scope for barrier and atomic operations.

Each invocation in a local workgroup must be in the same command scope instance.

Only task, mesh, and compute shaders have defined workgroups - other shader types cannot use workgroup functionality. For shaders that have defined workgroups, this set of invocations forms an invocation group as defined in the SPIR-V specification.

### 9.21.8. Subgroup

A subgroup (see the subsection “Control Flow” of section 2 of the SPIR-V 1.3 Revision 1 specification) is a set of invocations that can synchronize and share data with each other efficiently.

The Subgroup Scope can be used as both an Execution Scope and Memory Scope for barrier and atomic operations. Other subgroup features allow the use of group operations with subgroup scope.

If the shaderSubgroupClock feature is enabled, using the Subgroup Scope with the OpReadClockKHR instruction will read from a clock that is consistent across invocations in the same subgroup.

For shaders that have defined workgroups, each invocation in a subgroup must be in the same local workgroup.

In other shader stages, each invocation in a subgroup must be in the same device scope instance.

Only shader stages that support subgroup operations have defined subgroups.

### 9.21.9. Quad

A quad scope instance is formed of four shader invocations.

In a fragment shader, each invocation in a quad scope instance is formed of invocations in neighboring framebuffer locations \((x_i, y_i)\), where:

- \(i\) is the index of the invocation within the scope instance.
- \(w\) and \(h\) are the number of pixels the fragment covers in the \(x\) and \(y\) axes.
- \(w\) and \(h\) are identical for all participating invocations.
- \((x_0) = (x_1 - w) = (x_2) = (x_3 - w)\)
- \((y_0) = (y_1) = (y_2 - h) = (y_3 - h)\)

Each invocation has the same layer and sample indices.
In a compute shader, if the `DerivativeGroupQuadsNV` execution mode is specified, each invocation in a quad scope instance is formed of invocations with adjacent local invocation IDs \((x_i, y_i)\), where:

- \(i\) is the index of the invocation within the quad scope instance.
- \((x_0) = (x_1 - 1) = (x_2) = (x_3 - 1)\)
- \((y_0) = (y_1) = (y_2 - 1) = (y_3 - 1)\)
- \(x_0\) and \(y_0\) are integer multiples of 2.
- Each invocation has the same \(z\) coordinate.

In a compute shader, if the `DerivativeGroupLinearNV` execution mode is specified, each invocation in a quad scope instance is formed of invocations with adjacent local invocation indices \((l_i)\), where:

- \(i\) is the index of the invocation within the quad scope instance.
- \((l_0) = (l_1 - 1) = (l_2 - 2) = (l_3 - 3)\)
- \(l_0\) is an integer multiple of 4.

In all shaders, each invocation in a quad scope instance is formed of invocations in adjacent subgroup invocation indices \((s_i)\), where:

- \(i\) is the index of the invocation within the quad scope instance.
- \((s_0) = (s_1 - 1) = (s_2 - 2) = (s_3 - 3)\)
- \(s_0\) is an integer multiple of 4.

Each invocation in a quad scope instance must be in the same subgroup.

In a fragment shader, each invocation in a quad scope instance must be in the same primitive scope instance.

Fragment and compute shaders have defined quad scope instances. If the `quadOperationsInAllStages` limit is supported, any shader stages that support subgroup operations also have defined quad scope instances.

### 9.21.10. Fragment Interlock

A fragment interlock scope instance is formed of fragment shader invocations based on their framebuffer locations \((x,y,layer,sample)\), executed by commands inside a single subpass.

The specific set of invocations included varies based on the execution mode as follows:

- If the `SampleInterlockOrderedEXT` or `SampleInterlockUnorderedEXT` execution modes are used, only invocations with identical framebuffer locations \((x,y,layer,sample)\) are included.
- If the `PixelInterlockOrderedEXT` or `PixelInterlockUnorderedEXT` execution modes are used, fragments with different sample ids are also included.
- If the `ShadingRateInterlockOrderedEXT` or `ShadingRateInterlockUnorderedEXT` execution modes are used, fragments from neighbouring framebuffer locations are also included, as determined by the shading rate.
Only fragment shaders with one of the above execution modes have defined fragment interlock scope instances.

There is no specific Scope value for communication across invocations in a fragment interlock scope instance. However, this is implicitly used as a memory scope by `OpBeginInvocationInterlockEXT` and `OpEndInvocationInterlockEXT`.

Each invocation in a fragment interlock scope instance must be in the same queue family scope instance.

### 9.21.11. Invocation

The smallest scope is a single invocation; this is represented by the Invocation Scope in SPIR-V.

Fragment shader invocations must be in a primitive scope instance.

Invocations in fragment shaders that have a defined fragment interlock scope must be in a fragment interlock scope instance.

Invocations in shaders that have defined workgroups must be in a local workgroup.

Invocations in shaders that have a defined subgroup scope must be in a subgroup.

Invocations in shaders that have a defined quad scope must be in a quad scope instance.

All invocations in all stages must be in a command scope instance.

### 9.22. Group Operations

Group operations are executed by multiple invocations within a scope instance; with each invocation involved in calculating the result. This provides a mechanism for efficient communication between invocations in a particular scope instance.

Group operations all take a Scope defining the desired scope instance to operate within. Only the Subgroup scope can be used for these operations; the subgroupSupportedOperations limit defines which types of operation can be used.

#### 9.22.1. Basic Group Operations

Basic group operations include the use of `OpGroupNonUniformElect`, `OpControlBarrier`, `OpMemoryBarrier`, and atomic operations.

`OpGroupNonUniformElect` can be used to choose a single invocation to perform a task for the whole group. Only the invocation with the lowest id in the group will return `true`.

The Memory Model appendix defines the operation of barriers and atomics.

#### 9.22.2. Vote Group Operations

The vote group operations allow invocations within a group to compare values across a group. The
types of votes enabled are:

- Do all active group invocations agree that an expression is true?
- Do any active group invocations evaluate an expression to true?
- Do all active group invocations have the same value of an expression?

*Note*

These operations are useful in combination with control flow in that they allow for developers to check whether conditions match across the group and choose potentially faster code-paths in these cases.

### 9.22.3. Arithmetic Group Operations

The arithmetic group operations allow invocations to perform scans and reductions across a group. The operators supported are add, mul, min, max, and, or, xor.

For reductions, every invocation in a group will obtain the cumulative result of these operators applied to all values in the group. For exclusive scans, each invocation in a group will obtain the cumulative result of these operators applied to all values in invocations with a lower index in the group. Inclusive scans are identical to exclusive scans, except the cumulative result includes the operator applied to the value in the current invocation.

The order in which these operators are applied is implementation-dependent.

### 9.22.4. Ballot Group Operations

The ballot group operations allow invocations to perform more complex votes across the group. The ballot functionality allows all invocations within a group to provide a boolean value and get as a result what each invocation provided as their boolean value. The broadcast functionality allows values to be broadcast from an invocation to all other invocations within the group.

### 9.22.5. Shuffle Group Operations

The shuffle group operations allow invocations to read values from other invocations within a group.

### 9.22.6. Shuffle Relative Group Operations

The shuffle relative group operations allow invocations to read values from other invocations within the group relative to the current invocation in the group. The relative operations supported allow data to be shifted up and down through the invocations within a group.

### 9.22.7. Clustered Group Operations

The clustered group operations allow invocations to perform an operation among partitions of a group, such that the operation is only performed within the group invocations within a partition. The partitions for clustered group operations are consecutive power-of-two size groups of invocations and the cluster size **must** be known at pipeline creation time. The operations supported
are add, mul, min, max, and, or, xor.

9.23. Quad Group Operations

Quad group operations (OpGroupNonUniformQuad*) are a specialized type of group operations that only operate on quad scope instances. Whilst these instructions do include a Scope parameter, this scope is always overridden; only the quad scope instance is included in its execution scope.

Fragment shaders that statically execute quad group operations must launch sufficient invocations to ensure their correct operation; additional helper invocations are launched for framebuffer locations not covered by rasterized fragments if necessary.

The index used to select participating invocations is i, as described for a quad scope instance, defined as the quad index in the SPIR-V specification.

For OpGroupNonUniformQuadBroadcast this value is equal to Index. For OpGroupNonUniformQuadSwap, it is equal to the implicit Index used by each participating invocation.

9.24. Derivative Operations

Derivative operations calculate the partial derivative for an expression P as a function of an invocation’s x and y coordinates.

Derivative operations operate on a set of invocations known as a derivative group as defined in the SPIR-V specification. A derivative group is equivalent to the quad scope instance for a compute shader invocation, or the primitive scope instance for a fragment shader invocation.

Derivatives are calculated assuming that P is piecewise linear and continuous within the derivative group. All dynamic instances of explicit derivative instructions (OpDPdx*, OpDPdy*, and OpFwidth*) must be executed in control flow that is uniform within a derivative group. For other derivative operations, results are undefined if a dynamic instance is executed in control flow is not uniform within the derivative group.

Fragment shaders that statically execute derivative operations must launch sufficient invocations to ensure their correct operation; additional helper invocations are launched for framebuffer locations not covered by rasterized fragments if necessary.

Note

In a compute shader, it is the application’s responsibility to ensure that sufficient invocations are launched.

Derivative operations calculate their results as the difference between the result of P across invocations in the quad. For fine derivative operations (OpDPdxFine and OpDPdyFine), the values of DPdx(P) are calculated as

\[ DPdx(P_0) = DPdx(P_1) = P_1 - P_0 \]
\[ \text{DPdx}(P_2) = \text{DPdx}(P_3) = P_3 - P_2 \]

and the values of \( \text{DPdy}(P_i) \) are calculated as

\[ \begin{align*}
\text{DPdy}(P_0) &= \text{DPdy}(P_2) = P_2 - P_0 \\
\text{DPdy}(P_1) &= \text{DPdy}(P_3) = P_3 - P_1
\end{align*} \]

where \( i \) is the index of each invocation as described in *Quad*.

Coarse derivative operations (\text{OpDPdxCoarse} and \text{OpDPdyCoarse}), calculate their results in roughly the same manner, but \textbf{may} only calculate two values instead of four (one for each of \text{DPdx} and \text{DPdy}), reusing the same result no matter the originating invocation. If an implementation does this, it \textbf{should} use the fine derivative calculations described for \( P_0 \).

\( \text{Note} \)

Derivative values are calculated between fragments rather than pixels. If the fragment shader invocations involved in the calculation covers multiple pixels, these operations cover a wider area, resulting in larger derivative values. This in turn will result in a coarser level of detail being selected for image sampling operations using derivatives.

Applications may want to account for this when using multi-pixel fragments; if pixel derivatives are desired, applications should use explicit derivative operations and divide the results by the size of the fragment in each dimension as follows:

\[ \begin{align*}
\text{DPdx}(P_n)' &= \text{DPdx}(P_n) / w \\
\text{DPdy}(P_n)' &= \text{DPdy}(P_n) / h
\end{align*} \]

where \( w \) and \( h \) are the size of the fragments in the quad, and \( \text{DPdx}(P_n)' \) and \( \text{DPdy}(P_n)' \) are the pixel derivatives.

The results for \text{OpDPdx} and \text{OpDPdy} \textbf{may} be calculated as either fine or coarse derivatives, with implementations favouring the most efficient approach. Implementations \textbf{must} choose coarse or fine consistently between the two.

Executing \text{OpFwidthFine}, \text{OpFwidthCoarse}, or \text{OpFwidth} is equivalent to executing the corresponding \text{OpDPdx*} and \text{OpDPdy*} instructions, taking the absolute value of the results, and summing them.

Executing a \text{OpImage*Sample*ImplicitLod} instruction is equivalent to executing \text{OpDPdx(Coordinate)} and \text{OpDPdy(Coordinate)}, and passing the results as the \text{Grad} operands \( dx \) and \( dy \).
It is expected that using the `ImplicitLod` variants of sampling functions will be substantially more efficient than using the `ExplicitLod` variants with explicitly generated derivatives.

### 9.25. Helper Invocations

When performing derivative or quad group operations in a fragment shader, additional invocations may be spawned in order to ensure correct results. These additional invocations are known as helper invocations and can be identified by a non-zero value in the `HelperInvocation` built-in. Stores and atomic operations performed by helper invocations must not have any effect on memory, and values returned by atomic instructions in helper invocations are undefined.

Helper invocations may become inactive at any time for any reason, with one exception. If a helper invocation would be active if it were not a helper invocation, it must be active for derivative and quad group operations.

Helper invocations may become permanently inactive if all invocations in a quad scope instance become helper invocations.


A cooperative matrix type is a SPIR-V type where the storage for and computations performed on the matrix are spread across the invocations in a scope instance. These types give the implementation freedom in how to optimize matrix multiplies.

SPIR-V defines the types and instructions, but does not specify rules about what sizes/combinations are valid, and it is expected that different implementations may support different sizes.

To enumerate the supported cooperative matrix types and operations, call:

```c
// Provided by VK_NV_cooperative_matrix
VkResult vkGetPhysicalDeviceCooperativeMatrixPropertiesNV(
    VkPhysicalDevice                            physicalDevice,  
    uint32_t*                                   pPropertyCount,  
    VkCooperativeMatrixPropertiesNV*            pProperties);
```

- `physicalDevice` is the physical device.
- `pPropertyCount` is a pointer to an integer related to the number of cooperative matrix properties available or queried.
- `pProperties` is either `NULL` or a pointer to an array of `VkCooperativeMatrixPropertiesNV` structures.

If `pProperties` is `NULL`, then the number of cooperative matrix properties available is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of elements.

510 | Chapter 9. Shaders
structures actually written to \textit{pProperties}. If \textit{pPropertyCount} is less than the number of cooperative matrix properties available, at most \textit{pPropertyCount} structures will be written, and \textit{VK_INCOMPLETE} will be returned instead of \textit{VK_SUCCESS}, to indicate that not all the available cooperative matrix properties were returned.

### Valid Usage (Implicit)

- \textbf{VUID-vkGetPhysicalDeviceCooperativeMatrixPropertiesNV-physicalDevice-parameter}
  - \textit{physicalDevice} must be a valid \textit{VkPhysicalDevice} handle
- \textbf{VUID-vkGetPhysicalDeviceCooperativeMatrixPropertiesNV-pPropertyCount-parameter}
  - \textit{pPropertyCount} must be a valid pointer to a \textit{uint32_t} value
- \textbf{VUID-vkGetPhysicalDeviceCooperativeMatrixPropertiesNV-pProperties-parameter}
  - If the value referenced by \textit{pPropertyCount} is not 0, and \textit{pProperties} is not NULL, \textit{pProperties} must be a valid pointer to an array of \textit{pPropertyCount} \textit{VkCooperativeMatrixPropertiesNV} structures

### Return Codes

**Success**
- \textit{VK_SUCCESS}
- \textit{VK_INCOMPLETE}

**Failure**
- \textit{VK_ERROR_OUT_OF_HOST_MEMORY}
- \textit{VK_ERROR_OUT_OF_DEVICE_MEMORY}

Each \textit{VkCooperativeMatrixPropertiesNV} structure describes a single supported combination of types for a matrix multiply/add operation (\textit{OpCooperativeMatrixMulAddNV}). The multiply \textbf{can} be described in terms of the following variables and types (in SPIR-V pseudocode):

```plaintext
%A is of type OpTypeCooperativeMatrixNV %AType %scope %MSize %KSize
%B is of type OpTypeCooperativeMatrixNV %BType %scope %KSize %NSize
%C is of type OpTypeCooperativeMatrixNV % CType %scope %MSize %NSize
%D is of type OpTypeCooperativeMatrixNV %DType %scope %MSize %NSize

%D = %A * %B + %C // using OpCooperativeMatrixMulAddNV
```

A matrix multiply with these dimensions is known as an \textit{MxNxK} matrix multiply.

The \textit{VkCooperativeMatrixPropertiesNV} structure is defined as:
// Provided by VK_NV_cooperative_matrix
typedef struct VkCooperativeMatrixPropertiesNV {
    VkStructureType       sType;
    void*                 pNext;
    uint32_t              MSize;
    uint32_t              NSize;
    uint32_t              KSize;
    VkComponentTypeNV     AType;
    VkComponentTypeNV     BType;
    VkComponentTypeNV     CType;
    VkComponentTypeNV     DType;
    VkScopeNV             scope;
} VkCooperativeMatrixPropertiesNV;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **MSize** is the number of rows in matrices A, C, and D.
- **KSize** is the number of columns in matrix A and rows in matrix B.
- **NSize** is the number of columns in matrices B, C, D.
- **AType** is the component type of matrix A, of type \( \text{VkComponentTypeNV} \).
- **BType** is the component type of matrix B, of type \( \text{VkComponentTypeNV} \).
- ** CType** is the component type of matrix C, of type \( \text{VkComponentTypeNV} \).
- **DType** is the component type of matrix D, of type \( \text{VkComponentTypeNV} \).
- **scope** is the scope of all the matrix types, of type \( \text{VkScopeNV} \).

If some types are preferred over other types (e.g. for performance), they **should** appear earlier in the list enumerated by \( \text{vkGetPhysicalDeviceCooperativeMatrixPropertiesNV} \).

At least one entry in the list **must** have power of two values for all of **MSize**, **KSize**, and **NSize**.
Valid Usage (Implicit)

- **VUID-VkCooperativeMatrixPropertiesNV-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_COOPERATIVE_MATRIX_PROPERTIES_NV`

- **VUID-VkCooperativeMatrixPropertiesNV-pNext-pNext**
  - `pNext` must be `NULL`

- **VUID-VkCooperativeMatrixPropertiesNV-ATYPE-parameter**
  - `ATYPE` must be a valid `VkComponentTypeNV` value

- **VUID-VkCooperativeMatrixPropertiesNV-BTYPE-parameter**
  - `BTYPE` must be a valid `VkComponentTypeNV` value

- **VUID-VkCooperativeMatrixPropertiesNV-CTYPE-parameter**
  - `CTYPE` must be a valid `VkComponentTypeNV` value

- **VUID-VkCooperativeMatrixPropertiesNV-DTYPE-parameter**
  - `DTYPE` must be a valid `VkComponentTypeNV` value

- **VUID-VkCooperativeMatrixPropertiesNV-SCOPE-parameter**
  - `scope` must be a valid `VkScopeNV` value

Possible values for `VkScopeNV` include:

```c
// Provided by VK_NV_cooperative_matrix
typedef enum VkScopeNV {
    VK_SCOPE_DEVICE_NV = 1,
    VK_SCOPE_WORKGROUP_NV = 2,
    VK_SCOPE_SUBGROUP_NV = 3,
    VK_SCOPE_QUEUE_FAMILY_NV = 5,
} VkScopeNV;
```

- **VK_SCOPE_DEVICE_NV** corresponds to SPIR-V **Device** scope.
- **VK_SCOPE_WORKGROUP_NV** corresponds to SPIR-V **Workgroup** scope.
- **VK_SCOPE_SUBGROUP_NV** corresponds to SPIR-V **Subgroup** scope.
- **VK_SCOPE_QUEUE_FAMILY_NV** corresponds to SPIR-V **QueueFamily** scope.

All enum values match the corresponding SPIR-V value.

Possible values for `VkComponentTypeNV` include:
typedef enum VkComponentTypeNV {
    VK_COMPONENT_TYPE_FLOAT16_NV = 0,
    VK_COMPONENT_TYPE_FLOAT32_NV = 1,
    VK_COMPONENT_TYPE_FLOAT64_NV = 2,
    VK_COMPONENT_TYPE_SINT8_NV = 3,
    VK_COMPONENT_TYPE_SINT16_NV = 4,
    VK_COMPONENT_TYPE_SINT32_NV = 5,
    VK_COMPONENT_TYPE_SINT64_NV = 6,
    VK_COMPONENT_TYPE_UINT8_NV = 7,
    VK_COMPONENT_TYPE_UINT16_NV = 8,
    VK_COMPONENT_TYPE_UINT32_NV = 9,
    VK_COMPONENT_TYPE_UINT64_NV = 10,
} VkComponentTypeNV;

• VK_COMPONENT_TYPE_FLOAT16_NV corresponds to SPIR-V OpTypeFloat 16.
• VK_COMPONENT_TYPE_FLOAT32_NV corresponds to SPIR-V OpTypeFloat 32.
• VK_COMPONENT_TYPE_FLOAT64_NV corresponds to SPIR-V OpTypeFloat 64.
• VK_COMPONENT_TYPE_SINT8_NV corresponds to SPIR-V OpTypeInt 8 1.
• VK_COMPONENT_TYPE_SINT16_NV corresponds to SPIR-V OpTypeInt 16 1.
• VK_COMPONENT_TYPE_SINT32_NV corresponds to SPIR-V OpTypeInt 32 1.
• VK_COMPONENT_TYPE_SINT64_NV corresponds to SPIR-V OpTypeInt 64 1.
• VK_COMPONENT_TYPE_UINT8_NV corresponds to SPIR-V OpTypeInt 8 0.
• VK_COMPONENT_TYPE_UINT16_NV corresponds to SPIR-V OpTypeInt 16 0.
• VK_COMPONENT_TYPE_UINT32_NV corresponds to SPIR-V OpTypeInt 32 0.
• VK_COMPONENT_TYPE_UINT64_NV corresponds to SPIR-V OpTypeInt 64 0.

### 9.27. Validation Cache

Validation cache objects allow the result of internal validation to be reused, both within a single application run and between multiple runs. Reuse within a single run is achieved by passing the same validation cache object when creating supported Vulkan objects. Reuse across runs of an application is achieved by retrieving validation cache contents in one run of an application, saving the contents, and using them to preinitialize a validation cache on a subsequent run. The contents of the validation cache objects are managed by the validation layers. Applications can manage the host memory consumed by a validation cache object and control the amount of data retrieved from a validation cache object.

Validation cache objects are represented by VkValidationCacheEXT handles:
To create validation cache objects, call:

```cpp
// Provided by VK_EXT_validation_cache
VkResult vkCreateValidationCacheEXT(
    VkDevice                                    device,
    const VkValidationCacheCreateInfoEXT*       pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkValidationCacheEXT*                       pValidationCache);
```

- `device` is the logical device that creates the validation cache object.
- `pCreateInfo` is a pointer to a `VkValidationCacheCreateInfoEXT` structure containing the initial parameters for the validation cache object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pValidationCache` is a pointer to a `VkValidationCacheEXT` handle in which the resulting validation cache object is returned.

**Note**

Applications can track and manage the total host memory size of a validation cache object using the `pAllocator`. Applications can limit the amount of data retrieved from a validation cache object in `vkGetValidationCacheDataEXT`. Implementations should not internally limit the total number of entries added to a validation cache object or the total host memory consumed.

Once created, a validation cache can be passed to the `vkCreateShaderModule` command by adding this object to the `VkShaderModuleCreateInfo` structure’s `pNext` chain. If a `VkShaderModuleValidationCacheCreateInfoEXT` object is included in the `VkShaderModuleCreateInfo::pNext` chain, and its `validationCache` field is not `VK_NULL_HANDLE`, the implementation will query it for possible reuse opportunities and update it with new content. The use of the validation cache object in these commands is internally synchronized, and the same validation cache object can be used in multiple threads simultaneously.

**Note**

Implementations should make every effort to limit any critical sections to the actual accesses to the cache, which is expected to be significantly shorter than the duration of the `vkCreateShaderModule` command.
Valid Usage (Implicit)

- **VUID-vkCreateValidationCacheEXT-device-parameter**
  
  *device* **must** be a valid *VkDevice* handle

- **VUID-vkCreateValidationCacheEXT-pCreateInfo-parameter**
  
  *pCreateInfo* **must** be a valid pointer to a valid *VkValidationCacheCreateInfoEXT* structure

- **VUID-vkCreateValidationCacheEXT-pAllocator-parameter**
  
  If *pAllocator* is not **NULL**, *pAllocator* **must** be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkCreateValidationCacheEXT-pValidationCache-parameter**
  
  *pValidationCache* **must** be a valid pointer to a *VkValidationCacheEXT* handle

Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**

The *VkValidationCacheCreateInfoEXT* structure is defined as:

```c
// Provided by VK_EXT_validation_cache
typedef struct VkValidationCacheCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkValidationCacheCreateFlagsEXT flags;
    size_t initialDataSize;
    const void* pInitialData;
} VkValidationCacheCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **initialDataSize** is the number of bytes in *pInitialData*. If *initialDataSize* is zero, the validation cache will initially be empty.
- **pInitialData** is a pointer to previously retrieved validation cache data. If the validation cache data is incompatible (as defined below) with the device, the validation cache will be initially empty. If *initialDataSize* is zero, *pInitialData* is ignored.
Valid Usage

- VUID-VkValidationCacheCreateInfoEXT-initialDataSize-01534
  If `initialDataSize` is not 0, it must be equal to the size of `pInitialData`, as returned by `vkGetValidationCacheDataEXT` when `pInitialData` was originally retrieved.

- VUID-VkValidationCacheCreateInfoEXT-initialDataSize-01535
  If `initialDataSize` is not 0, `pInitialData` must have been retrieved from a previous call to `vkGetValidationCacheDataEXT`.

Valid Usage (Implicit)

- VUID-VkValidationCacheCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_VALIDATION_CACHE_CREATE_INFO_EXT`.

- VUID-VkValidationCacheCreateInfoEXT-pNext-pNext
  `pNext` must be `NULL`.

- VUID-VkValidationCacheCreateInfoEXT-flags-zerobitmask
  `flags` must be 0.

- VUID-VkValidationCacheCreateInfoEXT-pInitialData-parameter
  If `initialDataSize` is not 0, `pInitialData` must be a valid pointer to an array of `initialDataSize` bytes.

// Provided by VK_EXT_validation_cache
typedef VkFlags VkValidationCacheCreateFlagsEXT;

`VkValidationCacheCreateFlagsEXT` is a bitmask type for setting a mask, but is currently reserved for future use.

Validation cache objects can be merged using the command:

// Provided by VK_EXT_validation_cache
VkResult vkMergeValidationCachesEXT(
    VkDevice                                    device,  
    VkValidationCacheEXT                        dstCache,  
    uint32_t                                    srcCacheCount,  
    const VkValidationCacheEXT*                 pSrcCaches);

- `device` is the logical device that owns the validation cache objects.
- `dstCache` is the handle of the validation cache to merge results into.
- `srcCacheCount` is the length of the `pSrcCaches` array.
- `pSrcCaches` is a pointer to an array of validation cache handles, which will be merged into `dstCache`. The previous contents of `dstCache` are included after the merge.
Note
The details of the merge operation are implementation dependent, but implementations should merge the contents of the specified validation caches and prune duplicate entries.

Valid Usage
- VUID-vkMergeValidationCachesEXT-dstCache-01536
dstCache must not appear in the list of source caches

Valid Usage (Implicit)
- VUID-vkMergeValidationCachesEXT-device-parameter
device must be a valid VkDevice handle
- VUID-vkMergeValidationCachesEXT-dstCache-parameter
dstCache must be a valid VkValidationCacheEXT handle
- VUID-vkMergeValidationCachesEXT-pSrcCaches-parameter
pSrcCaches must be a valid pointer to an array of srcCacheCount valid VkValidationCacheEXT handles
- VUID-vkMergeValidationCachesEXT-srcCacheCount-arraylength
srcCacheCount must be greater than 0
- VUID-vkMergeValidationCachesEXT-dstCache-parent
dstCache must have been created, allocated, or retrieved from device
- VUID-vkMergeValidationCachesEXT-pSrcCaches-parent
Each element of pSrcCaches must have been created, allocated, or retrieved from device

Host Synchronization
- Host access to dstCache must be externally synchronized

Return Codes
Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Data can be retrieved from a validation cache object using the command:
Provided by VK_EXT_validation_cache

```c
VkResult vkGetValidationCacheDataEXT(
    VkDevice                                    device,
    VkValidationCacheEXT                        validationCache,
    size_t*                                     pDataSize,
    void*                                       pData);
```

- **device** is the logical device that owns the validation cache.
- **validationCache** is the validation cache to retrieve data from.
- **pDataSize** is a pointer to a value related to the amount of data in the validation cache, as described below.
- **pData** is either **NULL** or a pointer to a buffer.

If **pData** is **NULL**, then the maximum size of the data that can be retrieved from the validation cache, in bytes, is returned in **pDataSize**. Otherwise, **pDataSize** must point to a variable set by the user to the size of the buffer, in bytes, pointed to by **pData**, and on return the variable is overwritten with the amount of data actually written to **pData**. If **pDataSize** is less than the maximum size that can be retrieved by the validation cache, at most **pDataSize** bytes will be written to **pData**, and **vkGetValidationCacheDataEXT** will return **VK_INCOMPLETE** instead of **VK_SUCCESS**, to indicate that not all of the validation cache was returned.

Any data written to **pData** is valid and can be provided as the **pInitialData** member of the **VkValidationCacheCreateInfoEXT** structure passed to **vkCreateValidationCacheEXT**.

Two calls to **vkGetValidationCacheDataEXT** with the same parameters must retrieve the same data unless a command that modifies the contents of the cache is called between them.

Applications can store the data retrieved from the validation cache, and use these data, possibly in a future run of the application, to populate new validation cache objects. The results of validation, however, may depend on the vendor ID, device ID, driver version, and other details of the device. To enable applications to detect when previously retrieved data is incompatible with the device, the initial bytes written to **pData** must be a header consisting of the following members:

### Table 12. Layout for validation cache header version

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>length in bytes of the entire validation cache header written as a stream of bytes, with the least significant byte first</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>a <strong>VkValidationCacheHeaderVersionEXT</strong> value written as a stream of bytes, with the least significant byte first</td>
</tr>
<tr>
<td>8</td>
<td><strong>VK_UUID_SIZE</strong></td>
<td>a layer commit ID expressed as a UUID, which uniquely identifies the version of the validation layers used to generate these validation results</td>
</tr>
</tbody>
</table>
The first four bytes encode the length of the entire validation cache header, in bytes. This value includes all fields in the header including the validation cache version field and the size of the length field.

The next four bytes encode the validation cache version, as described for `VkValidationCacheHeaderVersionEXT`. A consumer of the validation cache should use the cache version to interpret the remainder of the cache header.

If `pDataSize` is less than what is necessary to store this header, nothing will be written to `pData` and zero will be written to `pDataSize`.

### Valid Usage (Implicit)

- **VUID-vkGetValidationCacheDataEXT-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkGetValidationCacheDataEXT-validationCache-parameter**
  
  `validationCache` must be a valid `VkValidationCacheEXT` handle

- **VUID-vkGetValidationCacheDataEXT-pDataSize-parameter**
  
  `pDataSize` must be a valid pointer to a `size_t` value

- **VUID-vkGetValidationCacheDataEXT-pData-parameter**
  
  If the value referenced by `pDataSize` is not 0, and `pData` is not NULL, `pData` must be a valid pointer to an array of `pDataSize` bytes

- **VUID-vkGetValidationCacheDataEXT-validationCache-parent**
  
  `validationCache` must have been created, allocated, or retrieved from `device`

### Return Codes

**Success**

- `VK_SUCCESS`
- `VK_INCOMPLETE`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Possible values of the second group of four bytes in the header returned by `vkGetValidationCacheDataEXT`, encoding the validation cache version, are:

```c
// Provided by VK_EXT_validation_cache
typedef enum VkValidationCacheHeaderVersionEXT {
    VK_VALIDATION_CACHE_HEADER_VERSION_ONE_EXT = 1,
} VkValidationCacheHeaderVersionEXT;
```
To destroy a validation cache, call:

```c
// Provided by VK_EXT_validation_cache
void vkDestroyValidationCacheEXT(
    VkDevice                                    device,
    VkValidationCacheEXT                        validationCache,
    const VkAllocationCallbacks*                pAllocator);
```

- `device` is the logical device that destroys the validation cache object.
- `validationCache` is the handle of the validation cache to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- **VUID-vkDestroyValidationCacheEXT-validationCache-01537**
  If `VkAllocationCallbacks` were provided when `validationCache` was created, a compatible set of callbacks must be provided here.

- **VUID-vkDestroyValidationCacheEXT-validationCache-01538**
  If no `VkAllocationCallbacks` were provided when `validationCache` was created, `pAllocator` must be `NULL`.

**Valid Usage (Implicit)**

- **VUID-vkDestroyValidationCacheEXT-device-parameter**
  `device` must be a valid `VkDevice` handle.

- **VUID-vkDestroyValidationCacheEXT-validationCache-parameter**
  If `validationCache` is not `VK_NULL_HANDLE`, `validationCache` must be a valid `VkValidationCacheEXT` handle.

- **VUID-vkDestroyValidationCacheEXT-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- **VUID-vkDestroyValidationCacheEXT-validationCache-parent**
  If `validationCache` is a valid handle, it must have been created, allocated, or retrieved from `device`.

**Host Synchronization**

- Host access to `validationCache` must be externally synchronized.
Chapter 10. Pipelines

The following figure shows a block diagram of the Vulkan pipelines. Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a graphics pipeline, a ray tracing pipeline, or a compute pipeline.

The graphics pipeline can be operated in two modes, as either primitive shading or mesh shading pipeline.

**Primitive Shading**

The first stage of the graphics pipeline (Input Assembler) assembles vertices to form geometric primitives such as points, lines, and triangles, based on a requested primitive topology. In the next stage (Vertex Shader) vertices can be transformed, computing positions and attributes for each vertex. If tessellation and/or geometry shaders are supported, they can then generate multiple primitives from a single input primitive, possibly changing the primitive topology or generating additional attribute data in the process.

**Mesh Shading**

When using the mesh shading pipeline input primitives are not assembled implicitly, but explicitly through the (Mesh Shader). The work on the mesh pipeline is initiated by the application drawing a set of mesh tasks.

If an optional (Task Shader) is active, each task triggers the execution of a task shader workgroup that will generate a new set of tasks upon completion. Each of these spawned tasks, or each of the original dispatched tasks if no task shader is present, triggers the execution of a mesh shader workgroup that produces an output mesh with a variable-sized number of primitives assembled from vertices stored in the output mesh.

**Common**

The final resulting primitives are clipped to a clip volume in preparation for the next stage, Rasterization. The rasterizer produces a series of fragments associated with a region of the framebuffer, from a two-dimensional description of a point, line segment, or triangle. These fragments are processed by fragment operations to determine whether generated values will be written to the framebuffer. fragment shading determines the values to be written to the framebuffer attachments. Framebuffer operations then read and write the color and depth/stencil attachments of the framebuffer for a given subpass of a render pass instance. The attachments can be used as input attachments in the fragment shader in a later subpass of the same render pass.

The compute pipeline is a separate pipeline from the graphics pipeline, which operates on one-, two-, or three-dimensional workgroups which can read from and write to buffer and image memory.

This ordering is meant only as a tool for describing Vulkan, not as a strict rule of how Vulkan is implemented, and we present it only as a means to organize the various operations of the pipelines.
Actual ordering guarantees between pipeline stages are explained in detail in the synchronization chapter.

Each pipeline is controlled by a monolithic object created from a description of all of the shader stages and any relevant fixed-function stages. Linking the whole pipeline together allows the optimization of shaders based on their input/outputs and eliminates expensive draw time state validation.

A pipeline object is bound to the current state using `vkCmdBindPipeline`. Any pipeline object state that is specified as dynamic is not applied to the current state when the pipeline object is bound, but is instead set by dynamic state setting commands.

No state, including dynamic state, is inherited from one command buffer to another.

Compute, ray tracing, and graphics pipelines are each represented by `VkPipeline` handles:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipeline)
```

### 10.1. Compute Pipelines

Compute pipelines consist of a single static compute shader stage and the pipeline layout.

The compute pipeline represents a compute shader and is created by calling `vkCreateComputePipelines` with `module` and `pName` selecting an entry point from a shader module, where that entry point defines a valid compute shader, in the `VkPipelineShaderStageCreateInfo` structure contained within the `VkComputePipelineCreateInfo` structure.

To create compute pipelines, call:

```cpp
""
// Provided by VK_VERSION_1_0
VkResult vkCreateComputePipelines(
    VkDevice                                    device,           
    VkPipelineCache                             pipelineCache,    
    uint32_t                                    createInfoCount, 
    const VkComputePipelineCreateInfo*          pCreateInfos,    
    const VkAllocationCallbacks*                pAllocator,       
    VkPipeline*                                 pPipelines);       

• device is the logical device that creates the compute pipelines.

• pipelineCache is either VK_NULL_HANDLE, indicating that pipeline caching is disabled; or the handle of a valid pipeline cache object, in which case use of that cache is enabled for the duration of the command.

• createInfoCount is the length of the pCreateInfos and pPipelines arrays.

• pCreateInfos is a pointer to an array of VkComputePipelineCreateInfo structures.

• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

• pPipelines is a pointer to an array of VkPipeline handles in which the resulting compute pipeline objects are returned.

Valid Usage

• VUID-vkCreateComputePipelines-flags-00695
  If the flags member of any element of pCreateInfos contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and the basePipelineIndex member of that same element is not -1, basePipelineIndex must be less than the index into pCreateInfos that corresponds to that element

• VUID-vkCreateComputePipelines-flags-00696
  If the flags member of any element of pCreateInfos contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, the base pipeline must have been created with the VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT flag set

• VUID-vkCreateComputePipelines-pipelineCache-02873
  If pipelineCache was created with VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT, host access to pipelineCache must be externally synchronized
Valid Usage (Implicit)

- VUID-vkCreateComputePipelines-device-parameter
  device must be a valid VkDevice handle

- VUID-vkCreateComputePipelines-pipelineCache-parameter
  If pipelineCache is not VK_NULL_HANDLE, pipelineCache must be a valid VkPipelineCache handle

- VUID-vkCreateComputePipelines-pCreateInfos-parameter
  pCreateInfos must be a valid pointer to an array of createInfoCount valid VkComputePipelineCreateInfo structures

- VUID-vkCreateComputePipelines-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateComputePipelines-pPipelines-parameter
  pPipelines must be a valid pointer to an array of createInfoCount VkPipeline handles

- VUID-vkCreateComputePipelines-createInfoCount-arraylength
  createInfoCount must be greater than 0

- VUID-vkCreateComputePipelines-pipelineCache-parent
  If pipelineCache is a valid handle, it must have been created, allocated, or retrieved from device

Return Codes

Success

- VK_SUCCESS
- VK_PIPELINE_COMPILE_REQUIRED_EXT

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INVALID_SHADER_NV

The VkComputePipelineCreateInfo structure is defined as:
// Provided by VK_VERSION_1_0

typedef struct VkComputePipelineCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkPipelineCreateFlags              flags;
    VkPipelineShaderStageCreateInfo    stage;
    VkPipelineLayout                   layout;
    VkPipeline                         basePipelineHandle;
    int32_t                            basePipelineIndex;
} VkComputePipelineCreateInfo;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is a bitmask of **VkPipelineCreateFlagBits** specifying how the pipeline will be generated.
• **stage** is a **VkPipelineShaderStageCreateInfo** structure describing the compute shader.
• **layout** is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
• **basePipelineHandle** is a pipeline to derive from
• **basePipelineIndex** is an index into the **pCreateInfos** parameter to use as a pipeline to derive from

The parameters **basePipelineHandle** and **basePipelineIndex** are described in more detail in **Pipeline Derivatives**.
Valid Usage

- **VUID-VkComputePipelineCreateInfo-flags-00697**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` must be a valid handle to a compute `VkPipeline`.

- **VUID-VkComputePipelineCreateInfo-flags-00698**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` must be a valid index into the calling command's `pCreateInfos` parameter.

- **VUID-VkComputePipelineCreateInfo-flags-00699**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not -1, `basePipelineHandle` must be `VK_NULL_HANDLE`.

- **VUID-VkComputePipelineCreateInfo-stage-00701**
  The `stage` member of `stage` must be `VK_SHADER_STAGE_COMPUTE_BIT`.

- **VUID-VkComputePipelineCreateInfo-stage-00702**
  The shader code for the entry point identified by `stage` and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Shader Interfaces chapter.

- **VUID-VkComputePipelineCreateInfo-layout-00703**
  `layout` must be consistent with the layout of the compute shader specified in `stage`.

- **VUID-VkComputePipelineCreateInfo-layout-01687**
  The number of resources in `layout` accessible to the compute shader stage must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`.

- **VUID-VkComputePipelineCreateInfo-flags-03364**
  `flags` must not include `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR`.

- **VUID-VkComputePipelineCreateInfo-flags-03365**
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR`.

- **VUID-VkComputePipelineCreateInfo-flags-03366**
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR`.

- **VUID-VkComputePipelineCreateInfo-flags-03367**
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR`.

- **VUID-VkComputePipelineCreateInfo-flags-03368**
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR`.

- **VUID-VkComputePipelineCreateInfo-flags-03369**
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR`.

- **VUID-VkComputePipelineCreateInfo-flags-03370**
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR`.
Valid Usage (Implicit)

- **VUID-VkComputePipelineCreateInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO`

- **VUID-VkComputePipelineCreateInfo-pNext-pNext**
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkPipelineCompilerControlCreateInfoAMD` or `VkPipelineCreationFeedbackCreateInfoEXT`.

- **VUID-VkComputePipelineCreateInfo-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkComputePipelineCreateInfo-flags-parameter**
  - `flags` must be a valid combination of `VkPipelineCreateFlagBits` values.

- **VUID-VkComputePipelineCreateInfo-stage-parameter**
  - `stage` must be a valid `VkPipelineShaderStageCreateInfo` structure.

- **VUID-VkComputePipelineCreateInfo-layout-parameter**
  - `layout` must be a valid `VkPipelineLayout` handle.

- **VUID-VkComputePipelineCreateInfo-commonparent**
  - Both of `basePipelineHandle`, and `layout` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

The `VkPipelineShaderStageCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineShaderStageCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineShaderStageCreateFlags    flags;
    VkShaderStageFlagBits               stage;
    VkShaderModule                      module;
    const char*                         pName;
    const VkSpecializationInfo*         pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **flags** is a bitmask of **VkPipelineShaderStageCreateFlagBits** specifying how the pipeline shader stage will be generated.

• **stage** is a **VkShaderStageFlagBits** value specifying a single pipeline stage.

• **module** is a **VkShaderModule** object containing the shader for this stage.

• **pName** is a pointer to a null-terminated UTF-8 string specifying the entry point name of the shader for this stage.

• **pSpecializationInfo** is a pointer to a **VkSpecializationInfo** structure, as described in **Specialization Constants**, or **NULL**.
Valid Usage

- VUID-VkPipelineShaderStageCreateInfo-stage-00704
  If the geometry shaders feature is not enabled, stage must not be `VK_SHADER_STAGE_GEOMETRY_BIT`

- VUID-VkPipelineShaderStageCreateInfo-stage-00705
  If the tessellation shaders feature is not enabled, stage must not be `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`

- VUID-VkPipelineShaderStageCreateInfo-stage-02091
  If the mesh shader feature is not enabled, stage must not be `VK_SHADER_STAGE_MESH_BIT_NV`

- VUID-VkPipelineShaderStageCreateInfo-stage-02092
  If the task shader feature is not enabled, stage must not be `VK_SHADER_STAGE_TASK_BIT_NV`

- VUID-VkPipelineShaderStageCreateInfo-maxClipDistances-00708
  If the identified entry point includes any variable in its interface that is declared with the ClipDistance BuiltIn decoration, that variable must not have an array size greater than `VkPhysicalDeviceLimits::maxClipDistances`

- VUID-VkPipelineShaderStageCreateInfo-maxCullDistances-00709
  If the identified entry point includes any variable in its interface that is declared with the CullDistance BuiltIn decoration, that variable must not have an array size greater than `VkPhysicalDeviceLimits::maxCullDistances`

- VUID-VkPipelineShaderStageCreateInfo-maxCombinedClipAndCullDistances-00710
  If the identified entry point includes any variables in its interface that are declared with the ClipDistance or CullDistance BuiltIn decoration, those variables must not have array sizes which sum to more than `VkPhysicalDeviceLimits::maxCombinedClipAndCullDistances`

- VUID-VkPipelineShaderStageCreateInfo-maxSampleMaskWords-00711
  If the identified entry point includes any variable in its interface that is declared with the SampleMask BuiltIn decoration, that variable must not have an array size greater than `VkPhysicalDeviceLimits::maxSampleMaskWords`

- VUID-VkPipelineShaderStageCreateInfo-stage-00712
  If stage is `VK_SHADER_STAGE_VERTEX_BIT`, the identified entry point must not include any input variable in its interface that is decorated with CullDistance

- VUID-VkPipelineShaderStageCreateInfo-stage-00713
  If stage is `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT` or `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, and the identified entry point has an OpExecutionMode instruction that specifies a patch size with OutputVertices, the patch size must be greater than 0 and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`
If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies a maximum output vertex count that is greater than 0 and less than or equal to `VkPhysicalDeviceLimits::maxGeometryOutputVertices`.

If `stage` is `VK_SHADER_STAGE_GEOMETRY_BIT`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies an invocation count that is greater than 0 and less than or equal to `VkPhysicalDeviceLimits::maxGeometryShaderInvocations`.

If `stage` is a vertex processing stage, and the identified entry point writes to `Layer` for any primitive, it **must** write the same value to `Layer` for all vertices of a given primitive.

If `stage` is a vertex processing stage, and the identified entry point writes to `ViewportIndex` for any primitive, it **must** write the same value to `ViewportIndex` for all vertices of a given primitive.

If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, the identified entry point **must** not include any output variables in its interface decorated with `CullDistance`.

If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, and the identified entry point writes to `FragDepth` in any execution path, it **must** write to `FragDepth` in all execution paths.

If `stage` is `VK_SHADER_STAGE_FRAGMENT_BIT`, and the identified entry point writes to `FragStencilRefEXT` in any execution path, it **must** write to `FragStencilRefEXT` in all execution paths.

If `stage` is `VK_SHADER_STAGE_MESH_BIT_NV`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies a maximum output vertex count, `OutputVertices`, that is greater than 0 and less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputVertices`.

If `stage` is `VK_SHADER_STAGE_MESH_BIT_NV`, the identified entry point **must** have an `OpExecutionMode` instruction that specifies a maximum output primitive count, `OutputPrimitivesNV`, that is greater than 0 and less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputPrimitives`.

If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set, the `subgroupSizeControl` feature **must** be enabled.

If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag set, the `computeFullSubgroups` feature **must** be enabled.

If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is included in...
the `pNext` chain, `flags` must not have the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set.

- **VUID-VkPipelineShaderStageCreateInfo-pNext-02755**

  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is included in the `pNext` chain, the `subgroupSizeControl` feature must be enabled, and `stage` must be a valid bit specified in `requiredSubgroupSizeStages`.

- **VUID-VkPipelineShaderStageCreateInfo-pNext-02756**

  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is included in the `pNext` chain and `stage` is `VK_SHADER_STAGE_COMPUTE_BIT`, the local workgroup size of the shader must be less than or equal to the product of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT::requiredSubgroupSize` and `maxComputeWorkgroupSubgroups`.

- **VUID-VkPipelineShaderStageCreateInfo-flags-02757**

  If a `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is included in the `pNext` chain, and `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag set, the local workgroup size in the X dimension of the pipeline must be a multiple of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT::requiredSubgroupSize`.

- **VUID-VkPipelineShaderStageCreateInfo-flags-02758**

  If `flags` has both the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` and `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flags set, the local workgroup size in the X dimension of the pipeline must be a multiple of `maxSubgroupSize`.

- **VUID-VkPipelineShaderStageCreateInfo-flags-02759**

  If `flags` has the `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` flag set and `flags` does not have the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set and no `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure is included in the `pNext` chain, the local workgroup size in the X dimension of the pipeline must be a multiple of `subgroupSize`.

- **VUID-VkPipelineShaderStageCreateInfo-module-04145**

  The SPIR-V code that was used to create `module` must be valid as described by the Khronos SPIR-V Specification after applying the specializations provided in `pSpecializationInfo`, if any, and then converting all specialization constants into fixed constants.
Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO`
- **pNext** must be `NULL` or a pointer to a valid instance of `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT`
- The **sType** value of each struct in the **pNext** chain must be unique
- **flags** must be a valid combination of `VkPipelineShaderStageCreateFlagBits` values
- **stage** must be a valid `VkShaderStageFlagBits` value
- **module** must be a valid `VkShaderModule` handle
- **pName** must be a null-terminated UTF-8 string
- If **pSpecializationInfo** is not `NULL`, **pSpecializationInfo** must be a valid pointer to a valid `VkSpecializationInfo` structure

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineShaderStageCreateFlags;
```

`VkPipelineShaderStageCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineShaderStageCreateFlagBits`.

Possible values of the **flags** member of `VkPipelineShaderStageCreateInfo` specifying how a pipeline shader stage is created, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineShaderStageCreateFlagBits {
    // Provided by VK_EXT_subgroup_size_control
    VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT = 0x00000001,
    // Provided by VK_EXT_subgroup_size_control
    VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT = 0x00000002,
} VkPipelineShaderStageCreateFlagBits;
```

- **VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT** specifies that the **SubgroupSize** may vary in the shader stage.
- **VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT** specifies that the subgroup sizes must be launched with all invocations active in the compute stage.
Note

If `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` and `VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT` are specified and `minSubgroupSize` does not equal `maxSubgroupSize` and no required subgroup size is specified, then the only way to guarantee that the 'X' dimension of the local workgroup size is a multiple of `SubgroupSize` is to make it a multiple of `maxSubgroupSize`. Under these conditions, you are guaranteed full subgroups but not any particular subgroup size.

Commands and structures which need to specify one or more shader stages do so using a bitmask whose bits correspond to stages. Bits which can be set to specify shader stages are:
typedef enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT = 0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS = 0x0000001F,
    VK_SHADER_STAGE_ALL = 0x7FFFFFFF,
    // Provided by VK_KHR_rasterization_state
    VK_SHADER_STAGE_RAYGEN_BIT_KHR = 0x00000100,
    // Provided by VK_KHR_rasterization_state
    VK_SHADER_STAGE_ANY_HIT_BIT_KHR = 0x00000200,
    // Provided by VK_KHR_rasterization_state
    VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR = 0x00000400,
    // Provided by VK_KHR_rasterization_state
    VK_SHADER_STAGE_MISS_BIT_KHR = 0x00000800,
    // Provided by VK_KHR_rasterization_state
    VK_SHADER_STAGE_INTERSECTION_BIT_KHR = 0x00001000,
    // Provided by VK_KHR_rasterization_state
    VK_SHADER_STAGE_CALLABLE_BIT_KHR = 0x00002000,
    // Provided by VK_NV_mesh_shader
    VK_SHADER_STAGE_TASK_BIT_NV = 0x00000040,
    // Provided by VK_NV_mesh_shader
    VK_SHADER_STAGE_MESH_BIT_NV = 0x00000080,
    // Provided by VK_NV_rasterization_state
    VK_SHADER_STAGE_RAYGEN_BIT_NV = VK_SHADER_STAGE_RAYGEN_BIT_KHR,
    // Provided by VK_NV_rasterization_state
    VK_SHADER_STAGE_ANY_HIT_BIT_NV = VK_SHADER_STAGE_ANY_HIT_BIT_KHR,
    // Provided by VK_NV_rasterization_state
    VK_SHADER_STAGE_CLOSEST_HIT_BIT_NV = VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR,
    // Provided by VK_NV_rasterization_state
    VK_SHADER_STAGE_MISS_BIT_NV = VK_SHADER_STAGE_MISS_BIT_KHR,
    // Provided by VK_NV_rasterization_state
    VK_SHADER_STAGE_INTERSECTION_BIT_NV = VK_SHADER_STAGE_INTERSECTION_BIT_KHR,
    // Provided by VK_NV_rasterization_state
    VK_SHADER_STAGE_CALLABLE_BIT_NV = VK_SHADER_STAGE_CALLABLE_BIT_KHR,
} VkShaderStageFlagBits;

- **VK_SHADER_STAGE_VERTEX_BIT** specifies the vertex stage.
- **VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT** specifies the tessellation control stage.
- **VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT** specifies the tessellation evaluation stage.
- **VK_SHADER_STAGE_GEOMETRY_BIT** specifies the geometry stage.
- **VK_SHADER_STAGE_FRAGMENT_BIT** specifies the fragment stage.
- **VK_SHADER_STAGE_COMPUTE_BIT** specifies the compute stage.
- **VK_SHADER_STAGE_ALL_GRAPHICS** is a combination of bits used as shorthand to specify all graphics.
stages defined above (excluding the compute stage).

- **VK_SHADER_STAGE_ALL** is a combination of bits used as shorthand to specify all shader stages supported by the device, including all additional stages which are introduced by extensions.
- **VK_SHADER_STAGE_TASK_BIT_NV** specifies the task stage.
- **VK_SHADER_STAGE_MESH_BIT_NV** specifies the mesh stage.
- **VK_SHADER_STAGE_RAYGEN_BIT_KHR** specifies the ray generation stage.
- **VK_SHADER_STAGE_ANY_HIT_BIT_KHR** specifies the any-hit stage.
- **VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR** specifies the closest hit stage.
- **VK_SHADER_STAGE_MISS_BIT_KHR** specifies the miss stage.
- **VK_SHADER_STAGE_INTERSECTION_BIT_KHR** specifies the intersection stage.
- **VK_SHADER_STAGE_CALLABLE_BIT_KHR** specifies the callable stage.

**Note**

**VK_SHADER_STAGE_ALL_GRAPHICS** only includes the original five graphics stages included in Vulkan 1.0, and not any stages added by extensions. Thus, it may not have the desired effect in all cases.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkShaderStageFlags;
```

**VkShaderStageFlags** is a bitmask type for setting a mask of zero or more **VkShaderStageFlagBits**.

The **VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_subgroup_size_control
typedef struct VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           requiredSubgroupSize;
} VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **requiredSubgroupSize** is an unsigned integer value that specifies the required subgroup size for the newly created pipeline shader stage.
Valid Usage

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-requiredSubgroupSize-02760
  \textit{requiredSubgroupSize} must be a power-of-two integer

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-requiredSubgroupSize-02761
  \textit{requiredSubgroupSize} must be greater or equal to \textit{minSubgroupSize}

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-requiredSubgroupSize-02762
  \textit{requiredSubgroupSize} must be less than or equal to \textit{maxSubgroupSize}

If a \texttt{VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT} structure is included in the \texttt{pNext} chain of \texttt{VkPipelineShaderStageCreateInfo}, it specifies that the pipeline shader stage being compiled has a required subgroup size.

Valid Usage (Implicit)

- VUID-VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT-sType-sType
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO_EXT}

10.2. Graphics Pipelines

Graphics pipelines consist of multiple shader stages, multiple fixed-function pipeline stages, and a pipeline layout.

To create graphics pipelines, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateGraphicsPipelines(
    VkDevice device,                     // Logical device
    VkPipelineCache pipelineCache,       // Pipeline cache
    uint32_t createInfoCount,            // Number of createInfos
    const VkGraphicsPipelineCreateInfo* pCreateInfo,    // Pointer to array of createInfos
    const VkAllocationCallbacks* pAllocator, // Controls host memory allocation
    VkPipeline* pPipelines);             // Pointer to array of pipelines
```

- \texttt{device} is the logical device that creates the graphics pipelines.
- \texttt{pipelineCache} is either \texttt{VK_NULL_HANDLE}, indicating that pipeline caching is disabled; or the handle of a valid pipeline cache object, in which case use of that cache is enabled for the duration of the command.
- \texttt{createInfoCount} is the length of the \texttt{pCreateInfos} and \texttt{pPipelines} arrays.
- \texttt{pCreateInfos} is a pointer to an array of \texttt{VkGraphicsPipelineCreateInfo} structures.
- \texttt{pAllocator} controls host memory allocation as described in the Memory Allocation chapter.
- \texttt{pPipelines} is a pointer to an array of \texttt{VkPipeline} handles in which the resulting graphics
pipeline objects are returned.

The \texttt{VkGraphicsPipelineCreateInfo} structure includes an array of shader create info structures containing all the desired active shader stages, as well as creation info to define all relevant fixed-function stages, and a pipeline layout.

### Valid Usage

- **VUID-vkCreateGraphicsPipelines-flags-00720**
  If the \texttt{flags} member of any element of \texttt{pCreateInfos} contains the \texttt{VK_PIPELINE_CREATE_DERIVATIVE_BIT} flag, and the \texttt{basePipelineIndex} member of that same element is not \texttt{-1}, \texttt{basePipelineIndex} must be less than the index into \texttt{pCreateInfos} that corresponds to that element.

- **VUID-vkCreateGraphicsPipelines-flags-00721**
  If the \texttt{flags} member of any element of \texttt{pCreateInfos} contains the \texttt{VK_PIPELINE_CREATE_DERIVATIVE_BIT} flag, the base pipeline must have been created with the \texttt{VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT} flag set.

- **VUID-vkCreateGraphicsPipelines-pipelineCache-02876**
  If \texttt{pipelineCache} was created with \texttt{VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT}, host access to \texttt{pipelineCache} must be externally synchronized.

### Note

An implicit cache may be provided by the implementation or a layer. For this reason, it is still valid to set \texttt{VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT} on \texttt{flags} for any element of \texttt{pCreateInfos} while passing \texttt{VK_NULL_HANDLE} for \texttt{pipelineCache}. 
Valid Usage (Implicit)

- VUID-vkCreateGraphicsPipelines-device-parameter
  
  *device** must be a valid **VkDevice** handle

- VUID-vkCreateGraphicsPipelines-pipelineCache-parameter
  
  If **pipelineCache** is not **VK_NULL_HANDLE**, **pipelineCache** must be a valid **VkPipelineCache** handle

- VUID-vkCreateGraphicsPipelines-pCreateInfos-parameter
  
  **pCreateInfos** must be a valid pointer to an array of **createInfoCount** valid **VkGraphicsPipelineCreateInfo** structures

- VUID-vkCreateGraphicsPipelines-pAllocator-parameter
  
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkCreateGraphicsPipelines-pPipelines-parameter
  
  **pPipelines** must be a valid pointer to an array of **createInfoCount** **VkPipeline** handles

- VUID-vkCreateGraphicsPipelines-createInfoCount-arraylength
  
  **createInfoCount** must be greater than **0**

- VUID-vkCreateGraphicsPipelines-pipelineCache-parent
  
  If **pipelineCache** is a valid handle, it must have been created, allocated, or retrieved from **device**

Return Codes

Success

- **VK_SUCCESS**
- **VK_PIPELINE_COMPILE_REQUIRED_EXT**

Failure

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
- **VK_ERROR_INVALID_SHADER_NV**

The **VkGraphicsPipelineCreateInfo** structure is defined as:
typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCreateFlags flags;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo* pStages;
    const VkPipelineVertexInputStateCreateInfo* pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo* pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo* pTessellationState;
    const VkPipelineViewportStateCreateInfo* pViewportState;
    const VkPipelineRasterizationStateCreateInfo* pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo* pMultisampleState;
    const VkPipelineColorBlendStateCreateInfo* pColorBlendState;
    const VkPipelineDynamicStateCreateInfo* pDynamicState;
    VkPipelineLayout layout;
    VkRenderPass renderPass;
    uint32_t subpass;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
} VkGraphicsPipelineCreateInfo;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkPipelineCreateFlagBits` specifying how the pipeline will be generated.
- `stageCount` is the number of entries in the `pStages` array.
- `pStages` is a pointer to an array of `stageCount` `VkPipelineShaderStageCreateInfo` structures describing the set of the shader stages to be included in the graphics pipeline.
- `pVertexInputState` is a pointer to a `VkPipelineVertexInputStateCreateInfo` structure. It is ignored if the pipeline includes a mesh shader stage.
- `pInputAssemblyState` is a pointer to a `VkPipelineInputAssemblyStateCreateInfo` structure which determines input assembly behavior, as described in Drawing Commands. It is ignored if the pipeline includes a mesh shader stage.
- `pTessellationState` is a pointer to a `VkPipelineTessellationStateCreateInfo` structure, and is ignored if the pipeline does not include a tessellation control shader stage and tessellation evaluation shader stage.
- `pViewportState` is a pointer to a `VkPipelineViewportStateCreateInfo` structure, and is ignored if the pipeline has rasterization disabled.
- `pRasterizationState` is a pointer to a `VkPipelineRasterizationStateCreateInfo` structure.
- `pMultisampleState` is a pointer to a `VkPipelineMultisampleStateCreateInfo` structure, and is ignored if the pipeline has rasterization disabled.
- `pDepthStencilState` is a pointer to a `VkPipelineDepthStencilStateCreateInfo` structure, and is ignored if the pipeline has rasterization disabled or if the subpass of the render pass the
pipeline is created against does not use a depth/stencil attachment.

- **pColorBlendState** is a pointer to a `VkPipelineColorBlendStateCreateInfo` structure, and is ignored if the pipeline has rasterization disabled or if the subpass of the render pass the pipeline is created against does not use any color attachments.

- **pDynamicState** is a pointer to a `VkPipelineDynamicStateCreateInfo` structure, and is used to indicate which properties of the pipeline state object are dynamic and can be changed independently of the pipeline state. This can be `NULL`, which means no state in the pipeline is considered dynamic.

- **layout** is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.

- **renderPass** is a handle to a render pass object describing the environment in which the pipeline will be used; the pipeline must only be used with an instance of any render pass compatible with the one provided. See Render Pass Compatibility for more information.

- **subpass** is the index of the subpass in the render pass where this pipeline will be used.

- **basePipelineHandle** is a pipeline to derive from.

- **basePipelineIndex** is an index into the `pCreateInfos` parameter to use as a pipeline to derive from.

The parameters **basePipelineHandle** and **basePipelineIndex** are described in more detail in Pipeline Derivatives.

If any shader stage fails to compile, the compile log will be reported back to the application, and **VK_ERROR_INVALID_SHADER_NV** will be generated.
Valid Usage

- **VUID-VkGraphicsPipelineCreateInfo-flags-00722**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` must be a valid handle to a graphics `VkPipeline`

- **VUID-VkGraphicsPipelineCreateInfo-flags-00723**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` must be a valid index into the calling command’s `pCreateInfo` parameter.

- **VUID-VkGraphicsPipelineCreateInfo-flags-00724**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` must be -1.

- **VUID-VkGraphicsPipelineCreateInfo-stage-00726**
  The `stage` member of each element of `pStages` must be unique.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-02095**
  The geometric shader stages provided in `pStages` must be either from the mesh shading pipeline (`stage` is `VK_SHADER_STAGE_TASK_BIT_NV` or `VK_SHADER_STAGE_MESH_BIT_NV`) or from the primitive shading pipeline (`stage` is `VK_SHADER_STAGE_VERTEX_BIT`, `VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT`, `VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT`, or `VK_SHADER_STAGE_GEOMETRY_BIT`).

- **VUID-VkGraphicsPipelineCreateInfo-stage-02096**
  The `stage` member of one element of `pStages` must be either `VK_SHADER_STAGE_VERTEX_BIT` or `VK_SHADER_STAGE_MESH_BIT_NV`.

- **VUID-VkGraphicsPipelineCreateInfo-stage-00728**
  The `stage` member of each element of `pStages` must not be `VK_SHADER_STAGE_COMPUTE_BIT`.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00729**
  If `pStages` includes a tessellation control shader stage, it must include a tessellation evaluation shader stage.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00730**
  If `pStages` includes a tessellation evaluation shader stage, it must include a tessellation control shader stage.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00731**
  If `pStages` includes a tessellation control shader stage and a tessellation evaluation shader stage, `pTessellationState` must be a valid pointer to a valid `VkPipelineTessellationStateCreateInfo` structure.

- **VUID-VkGraphicsPipelineCreateInfo-pStages-00732**
  If `pStages` includes tessellation shader stages, the shader code of at least one stage must contain an `OpExecutionMode` instruction that specifies the type of subdivision in the pipeline.
• VUID-VkGraphicsPipelineCreateInfo-pStages-00733
If `pStages` includes tessellation shader stages, and the shader code of both stages contain an `OpExecutionMode` instruction that specifies the type of subdivision in the pipeline, they **must** both specify the same subdivision mode.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00734
If `pStages` includes tessellation shader stages, the shader code of at least one stage **must** contain an `OpExecutionMode` instruction that specifies the output patch size in the pipeline.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00735
If `pStages` includes tessellation shader stages, and the shader code of both contain an `OpExecutionMode` instruction that specifies the output patch size in the pipeline, they **must** both specify the same patch size.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00736
If `pStages` includes tessellation shader stages, the `topology` member of `pInputAssembly` **must** be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`.

• VUID-VkGraphicsPipelineCreateInfo-topology-00737
If the `topology` member of `pInputAssembly` is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, `pStages` **must** include tessellation shader stages.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00738
If `pStages` includes a geometry shader stage, and does not include any tessellation shader stages, its shader code **must** contain an `OpExecutionMode` instruction that specifies an input primitive type that is compatible with the primitive topology specified in `pInputAssembly`.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00739
If `pStages` includes a geometry shader stage, and also includes tessellation shader stages, its shader code **must** contain an `OpExecutionMode` instruction that specifies an input primitive type that is compatible with the primitive topology that is output by the tessellation stages.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00740
If `pStages` includes a fragment shader stage and a geometry shader stage, and the fragment shader code reads from an input variable that is decorated with `PrimitiveID`, then the geometry shader code **must** write to a matching output variable, decorated with `PrimitiveID`, in all execution paths.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00741
If `pStages` includes a fragment shader stage, its shader code **must** not read from any input attachment that is defined as `VK_ATTACHMENT_UNUSED` in `subpass`.

• VUID-VkGraphicsPipelineCreateInfo-pStages-00742
The shader code for the entry points identified by `pStages`, and the rest of the state identified by this structure **must** adhere to the pipeline linking rules described in the **Shader Interfaces** chapter.

• VUID-VkGraphicsPipelineCreateInfo-subpass-01756
If rasterization is not disabled and `subpass` uses a depth/stencil attachment in `renderPass` that has a layout of `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL` in the `VkAttachmentReference` defined by `subpass`, the `depthWriteEnable` member of `pDepthStencilState` **must** be `VK_FALSE`.
If rasterization is not disabled and subpass uses a depth/stencil attachment in renderPass that has a layout of `VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL` or `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL` in the VkAttachmentReference defined by subpass, the failOp, passOp and depthFailOp members of each of the front and back members of pDepthStencilState must be `VK_STENCIL_OP_KEEP`.

If rasterization is not disabled, then for each color attachment in the subpass, if the potential format features of the format of the corresponding attachment description do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be `VK_FALSE`.

If rasterization is not disabled and the subpass uses color attachments, the attachmentCount member of pColorBlendState must be equal to the colorAttachmentCount used to create subpass.

If no element of the pDynamicStates member of pDynamicState is `VK_DYNAMIC_STATE_VIEWPORT` or `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT`, the pViewports member of pViewportState must be a valid pointer to an array of pViewportState->viewportCount valid VkViewport structures.

If no element of the pDynamicStates member of pDynamicState is `VK_DYNAMIC_STATE_SCISSOR` or `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT`, the pScissors member of pViewportState must be a valid pointer to an array of pViewportState->scissorCount VkRect2D structures.

If the wide lines feature is not enabled, and no element of the pDynamicStates member of pDynamicState is `VK_DYNAMIC_STATE_LINE_WIDTH`, the lineWidth member of pRasterizationState must be 1.0.

If the rasterizerDiscardEnable member of pRasterizationState is `VK_FALSE`, or if the graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled, pViewportState must be a valid pointer to a valid VkPipelineViewportStateCreateInfo structure.

If the rasterizerDiscardEnable member of pRasterizationState is `VK_FALSE`, or if the graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled, pMultisampleState must be a valid pointer to a valid VkPipelineMultisampleStateCreateInfo structure.

If the rasterizerDiscardEnable member of pRasterizationState is `VK_FALSE`, or if the graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled, and subpass uses a depth/stencil attachment, pDepthStencilState must be a valid pointer to a valid
If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, or if the graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled, and `subpass` uses color attachments, `pColorBlendState` must be a valid pointer to a valid `VkPipelineColorBlendStateCreateInfo` structure.

If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, or if the graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled, `pColorBlendState->attachmentCount` must be greater than the index of all color attachments that are not `VK_ATTACHMENT_UNUSED` for the `subpass` index in `renderPass`.

If the depth bias clamping feature is not enabled, no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BIAS`, and the `depthBiasEnable` member of `pRasterizationState` is `VK_TRUE`, the `depthBiasClamp` member of `pRasterizationState` must be 0.0.

If the `VK_EXT_depth_range_unrestricted` extension is not enabled and no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DEPTH_BOUNDS`, and the `depthBoundsTestEnable` member of `pDepthStencilState` is `VK_TRUE`, the `minDepthBounds` and `maxDepthBounds` members of `pDepthStencilState` must be between 0.0 and 1.0, inclusive.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`, and the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure included in the `pNext` chain of `pMultisampleState` is `VK_TRUE`, `sampleLocationsInfo.sampleLocationGridSize.width` must evenly divide `VkMultisamplePropertiesEXT::sampleLocationGridSize.width` as returned by `vkGetPhysicalDeviceMultisamplePropertiesEXT` with a `samples` parameter equaling `rasterizationSamples`.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`, and the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure included in the `pNext` chain of `pMultisampleState` is `VK_TRUE`, `sampleLocationsInfo.sampleLocationGridSize.height` must evenly divide `VkMultisamplePropertiesEXT::sampleLocationGridSize.height` as returned by `vkGetPhysicalDeviceMultisamplePropertiesEXT` with a `samples` parameter equaling `rasterizationSamples`.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`, and the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure included in the `pNext` chain of `pMultisampleState` is `VK_TRUE`, `sampleLocationsInfo.sampleLocationsPerPixel` must equal `rasterizationSamples`.
If the `sampleLocationsEnable` member of a `VkPipelineSampleLocationsStateCreateInfoEXT` structure included in the `pNext` chain of `pMultisampleState` is `VK_TRUE`, the fragment shader code must not statically use the extended instruction `InterpolateAtSample`.

*layout* must be consistent with all shaders specified in `pStages`.

If neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, and if `subpass` uses color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` must be the same as the sample count for those subpass attachments.

If the `VK_AMD_mixed_attachment_samples` extension is enabled, and if `subpass` uses color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` must equal the maximum of the sample counts of those subpass attachments.

If the `VK_NV_framebuffer_mixed_samples` extension is enabled, and if `subpass` has a depth/stencil attachment and depth test, stencil test, or depth bounds test are enabled, then the `rasterizationSamples` member of `pMultisampleState` must be the same as the sample count of the depth/stencil attachment.

If the `VK_NV_framebuffer_mixed_samples` extension is enabled, and if `subpass` has any color attachments, then the `rasterizationSamples` member of `pMultisampleState` must be greater than or equal to the sample count for those subpass attachments.

If the `VK_NV_coverage_reduction_mode` extension is enabled, the coverage reduction mode specified by `VkPipelineCoverageReductionStateCreateInfoNV::coverageReductionMode`, the `rasterizationSamples` member of `pMultisampleState` and the sample counts for the color and depth/stencil attachments (if the subpass has them) must be a valid combination returned by `vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV`.

If `subpass` does not use any color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` must follow the rules for a zero-attachment subpass.

*subpass* must be a valid subpass within `renderPass`.

If the `renderPass` has multiview enabled and `subpass` has more than one bit set in the view mask and `multiviewTessellationShader` is not enabled, then `pStages` must not include tessellation shaders.

If the `renderPass` has multiview enabled and `subpass` has more than one bit set in the view mask and `multiviewGeometryShader` is not enabled, then `pStages` must not include a
If the `renderPass` has multiview enabled and `subpass` has more than one bit set in the view mask, shaders in the pipeline must not write to the `Layer` built-in output.

If the `renderPass` has multiview enabled, then all shaders must not include variables decorated with the `Layer` built-in decoration in their interfaces.

Flags must not contain the `VK_PIPELINE_CREATE_DISPATCH_BASE` flag.

If `pStages` includes a fragment shader stage and an input attachment was referenced by an `aspectMask` at `renderPass` creation time, its shader code must only read from the aspects that were specified for that input attachment.

The number of resources in `layout` accessible to each shader stage that is used by the pipeline must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV`, and the `viewportWScalingEnable` member of a `VkPipelineViewportWScalingStateCreateInfoNV` structure, included in the `pNext` chain of `pViewportState`, is `VK_TRUE`, the `pViewportWScalings` member of the `VkPipelineViewportWScalingStateCreateInfoNV` must be a pointer to an array of `VkPipelineViewportWScalingStateCreateInfoNV::viewportCount` valid `VkViewportWScalingNV` structures.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV`, and if `pViewportState->pNext` chain includes a `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure, and if its `exclusiveScissorCount` member is not 0, then its `pExclusiveScissors` member must be a valid pointer to an array of `exclusiveScissorCount` `VkRect2D` structures.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV`, and if `pViewportState->pNext` chain includes a `VkPipelineViewportShadingRateImageStateCreateInfoNV` structure, then its `pShadingRatePalettes` member must be a valid pointer to an array of `viewportCount` valid `VkShadingRatePaletteNV` structures.

If no element of the `pDynamicStates` member of `pDynamicState` is `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT`, and if `pNext` chain includes a `VkPipelineDiscardRectangleStateCreateInfoEXT` structure, and if its `discardRectangleCount` member is not 0, then its `pDiscardRectangles` member must be a valid pointer to an array of `discardRectangleCount` `VkRect2D` structures.

If `pStages` includes a vertex shader stage, `pVertexInputState` must be a valid pointer to a
valid \texttt{VkPipelineVertexInputStateCreateInfo} structure

- VUID-VkGraphicsPipelineCreateInfo-pStages-02098
  If \texttt{pStages} includes a vertex shader stage, \texttt{pInputAssemblyState} must be a valid pointer to a valid \texttt{VkPipelineInputAssemblyStateCreateInfo} structure

- VUID-VkGraphicsPipelineCreateInfo-pStages-02317
  The \texttt{Xfb} execution mode can be specified by only one shader stage in \texttt{pStages}

- VUID-VkGraphicsPipelineCreateInfo-pStages-02318
  If any shader stage in \texttt{pStages} specifies \texttt{Xfb} execution mode it must be the last vertex processing stage

- VUID-VkGraphicsPipelineCreateInfo-rasterizationStream-02319
  If a \texttt{VkPipelineRasterizationStateStreamCreateInfoEXT::rasterizationStream} value other than zero is specified, all variables in the output interface of the entry point being compiled decorated with \texttt{Position}, \texttt{PointSize}, \texttt{ClipDistance}, or \texttt{CullDistance} must be decorated with identical \texttt{Stream} values that match the \texttt{rasterizationStream}

- VUID-VkGraphicsPipelineCreateInfo-rasterizationStream-02320
  If \texttt{VkPipelineRasterizationStateStreamCreateInfoEXT::rasterizationStream} is zero, or not specified, all variables in the output interface of the entry point being compiled decorated with \texttt{Position}, \texttt{PointSize}, \texttt{ClipDistance}, or \texttt{CullDistance} must be decorated with a \texttt{Stream} value of zero, or must not specify the \texttt{Stream} decoration

- VUID-VkGraphicsPipelineCreateInfo-geometryStreams-02321
  If the last vertex processing stage is a geometry shader, and that geometry shader uses the \texttt{GeometryStreams} capability, then \texttt{VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams feature} must be enabled

- VUID-VkGraphicsPipelineCreateInfo-None-02322
  If there are any mesh shader stages in the pipeline there must not be any shader stage in the pipeline with a \texttt{Xfb} execution mode

- VUID-VkGraphicsPipelineCreateInfo-lineRasterizationMode-02766
  If the \texttt{lineRasterizationMode} member of a \texttt{VkPipelineRasterizationLineStateCreateInfoEXT} structure included in the \texttt{pNext} chain of \texttt{pRasterizationState} is \texttt{VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT} or \texttt{VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT} and if rasterization is enabled, then the \texttt{alphaToCoverageEnable}, \texttt{alphaToOneEnable}, and \texttt{sampleShadingEnable} members of \texttt{pMultisampleState} must all be \texttt{VK_FALSE}

- VUID-VkGraphicsPipelineCreateInfo-stippledLineEnable-02767
  If the \texttt{stippledLineEnable} member of \texttt{VkPipelineRasterizationLineStateCreateInfoEXT} is \texttt{VK_TRUE} and no element of the \texttt{pDynamicStates} member of \texttt{pDynamicState} is \texttt{VK_DYNAMIC_STATE_LINE_STIPPLE_EXT}, then the \texttt{lineStippleFactor} member of \texttt{VkPipelineRasterizationLineStateCreateInfoEXT} must be in the range \([1,256]\)

- VUID-VkGraphicsPipelineCreateInfo-flags-03371
  \texttt{flags} must not include \texttt{VK_PIPELINE_CREATE_LIBRARY_BIT_KHR}

- VUID-VkGraphicsPipelineCreateInfo-flags-03372
  \texttt{flags} must not include \texttt{VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR}

- VUID-VkGraphicsPipelineCreateInfo-flags-03373
  \texttt{flags} must not include \texttt{VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR}
flags **must not include** VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR

- VUID-VkGraphicsPipelineCreateInfo-flags-03374
- VUID-VkGraphicsPipelineCreateInfo-flags-03375
- VUID-VkGraphicsPipelineCreateInfo-flags-03376
- VUID-VkGraphicsPipelineCreateInfo-flags-03377
- VUID-VkGraphicsPipelineCreateInfo-flags-03378
- VUID-VkGraphicsPipelineCreateInfo-flags-03379
- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04132
- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04133
- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04868
- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04869
- VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04870

If the **extendedDynamicState** feature is not enabled, then **must** be no element of the pDynamicStates member of pDynamicState set to

- VK_DYNAMIC_STATE_CULL_MODE_EXT, VK_DYNAMIC_STATE_FRONT_FACE_EXT, VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT,
- VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT, VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT,
- VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT, VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT, VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT,
- VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT, VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT,
- VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT, or VK_DYNAMIC_STATE_STENCIL_OP_EXT

If **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** is included in the pDynamicStates array then **viewportCount must be zero**

If **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT** is included in the pDynamicStates array then **scissorCount must be zero**

If **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** is included in the pDynamicStates array then **VK_DYNAMIC_STATE_VIEWPORT must not be present**

If **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT** is included in the pDynamicStates array then **VK_DYNAMIC_STATE_SCISSOR must not be present**

If the **extendedDynamicState2** feature is not enabled, then **must** be no element of the pDynamicStates member of pDynamicState set to

- VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT, VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT, or
- VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT

If the **extendedDynamicState2LogicOp** feature is not enabled, then **must** be no element of the pDynamicStates member of pDynamicState set to **VK_DYNAMIC_STATE_LOGIC_OP_EXT**

Chapter 10. Pipelines | 549
If the extendedDynamicState2PatchControlPoints feature is not enabled, there must be no element of the pDynamicStates member of pDynamicState set to VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT.

- **VUID-VkGraphicsPipelineCreateInfo-flags-02877**
  
  If flags includes VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV, then the VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands feature must be enabled.

- **VUID-VkGraphicsPipelineCreateInfo-flags-02966**
  
  If flags includes VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV, then all stages must not specify Xfb execution mode.

- **VUID-VkGraphicsPipelineCreateInfo-pipelineCreationCacheControl-02878**
  
  If the pipelineCreationCacheControl feature is not enabled, flags must not include VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT or VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04494**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width must be greater than or equal to 1.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04495**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height must be greater than or equal to 1.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04496**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width must be a power-of-two value.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04497**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height must be a power-of-two value.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04498**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width must be less than or equal to 4.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04499**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height must be less than or equal to 4.

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicState-04500**
  
  If VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR is not included in pDynamicState->pDynamicStates, and the pipelineFragmentShadingRate feature is not enabled, VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.width and VkPipelineFragmentShadingRateStateCreateInfoKHR::fragmentSize.height must both be equal to 1.
If \( \text{VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR} \) is not included in \( \text{pDynamicState} \), and the \text{primitiveFragmentShadingRate} feature is not enabled, \( \text{vkPipelineFragmentShadingRateStateCreateInfoKHR} :: \text{combinerOps}[0] \) must be \( \text{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR} \).

If \( \text{VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR} \) is not included in \( \text{pDynamicState} \), and the \text{attachmentFragmentShadingRate} feature is not enabled, \( \text{vkPipelineFragmentShadingRateStateCreateInfoKHR} :: \text{combinerOps}[1] \) must be \( \text{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR} \).

If the \text{primitiveFragmentShadingRateWithMultipleViewports} limit is not supported, \( \text{VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT} \) is not included in \( \text{pDynamicState} \), and \( \text{VkPipelineViewportStateCreateInfo} :: \text{viewportCount} \) is greater than 1, entry points specified in \( \text{pStages} \) must not write to the \text{PrimitiveShadingRateKHR} built-in.

If the \text{primitiveFragmentShadingRateWithMultipleViewports} limit is not supported, and entry points specified in \( \text{pStages} \) write to the \text{ViewportIndex} built-in, they must not also write to the \text{PrimitiveShadingRateKHR} built-in.

If the \text{primitiveFragmentShadingRateWithMultipleViewports} limit is not supported, and entry points specified in \( \text{pStages} \) write to the \text{ViewportMaskNV} built-in, they must not also write to the \text{PrimitiveShadingRateKHR} built-in.

If the \text{fragmentShadingRateNonTrivialCombinerOps} limit is not supported and \( \text{VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR} \) is not included in \( \text{pDynamicState} \), elements of \( \text{vkPipelineFragmentShadingRateStateCreateInfoKHR} :: \text{combinerOps} \) must be \( \text{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR} \) or \( \text{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR} \).

If \( \text{VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR} \) is not included in \( \text{pDynamicState} \), and the \text{fragmentShadingRateEnums} feature is not enabled, \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV} :: \text{shadingRateType} \) must be equal to \( \text{VK_FRAGMENT_SHADING_RATE_TYPE_FRAGMENT_SIZE_NV} \).

If \( \text{VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR} \) is not included in \( \text{pDynamicState} \), and the \text{pipelineFragmentShadingRate} feature is not enabled, \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV} :: \text{shadingRate} \) must be equal to \( \text{VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_PIXEL_NV} \).

If \( \text{VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR} \) is not included in \( \text{pDynamicState} \), and the \text{primitiveFragmentShadingRate} feature is not enabled, \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV} :: \text{combinerOps}[0] \) must be \( \text{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR} \).
If \( \text{VK\_DYNAMIC\_STATE\_FRAGMENT\_SHADING\_RATE\_KHR} \) is not included in \( p\text{DynamicStates} \), and the \text{attachmentFragmentShadingRate} feature is not enabled, \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV}::\text{combinerOps}[1] \) \textbf{must} be \( \text{VK\_FRAGMENT\_SHADING\_RATE\_COMBINER\_OP\_KEEP\_KHR} \).

- **VUID-VkGraphicsPipelineCreateInfo-fragmentShadingRateNonTrivialCombinerOps-04573**
  If the \text{fragmentShadingRateNonTrivialCombinerOps} limit is not supported and \( \text{VK\_DYNAMIC\_STATE\_FRAGMENT\_SHADING\_RATE\_KHR} \) is not included in \( p\text{DynamicStates} \), elements of \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV}::\text{combinerOps} \) \textbf{must} be \( \text{VK\_FRAGMENT\_SHADING\_RATE\_COMBINER\_OP\_KEEP\_KHR} \) or \( \text{VK\_FRAGMENT\_SHADING\_RATE\_COMBINER\_OP\_REPLACE\_KHR} \).

- **VUID-VkGraphicsPipelineCreateInfo-None-04574**
  If the \text{supersampleFragmentShadingRates} feature is not enabled, \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV}::\text{shadingRate} \) \textbf{must} not be equal to \( \text{VK\_FRAGMENT\_SHADING\_RATE\_2\_INVOCATIONS\_PER\_PIXEL\_NV} \), \( \text{VK\_FRAGMENT\_SHADING\_RATE\_4\_INVOCATIONS\_PER\_PIXEL\_NV} \), \( \text{VK\_FRAGMENT\_SHADING\_RATE\_8\_INVOCATIONS\_PER\_PIXEL\_NV} \), or \( \text{VK\_FRAGMENT\_SHADING\_RATE\_16\_INVOCATIONS\_PER\_PIXEL\_NV} \).

- **VUID-VkGraphicsPipelineCreateInfo-None-04575**
  If the \text{noInvocationFragmentShadingRates} feature is not enabled, \( \text{VkPipelineFragmentShadingRateEnumStateCreateInfoNV}::\text{shadingRate} \) \textbf{must} not be equal to \( \text{VK\_FRAGMENT\_SHADING\_RATE\_NO\_INVOCATIONS\_NV} \).

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-03578**
  All elements of the \( p\text{DynamicStates} \) member of \( p\text{DynamicState} \) \textbf{must} not be \( \text{VK\_DYNAMIC\_STATE\_RAY\_TRACING\_PIPELINE\_STACK\_SIZE\_KHR} \).

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04807**
  If the \text{vertexInputDynamicState} feature is not enabled, there \textbf{must} be no element of the \( p\text{DynamicStates} \) member of \( p\text{DynamicState} \) set to \( \text{VK\_DYNAMIC\_STATE\_VERTEX\_INPUT\_EXT} \).

- **VUID-VkGraphicsPipelineCreateInfo-pDynamicStates-04808**
  If the \text{colorWriteEnable} feature is not enabled, there \textbf{must} be no element of the \( p\text{DynamicStates} \) member of \( p\text{DynamicState} \) set to \( \text{VK\_DYNAMIC\_STATE\_COLOR\_WRITE\_ENABLE\_EXT} \).
Valid Usage (Implicit)

• VUID-VkGraphicsPipelineCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO

• VUID-VkGraphicsPipelineCreateInfo-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkGraphicsPipelineShaderGroupsCreateInfoNV, VkPipelineCompilerControlCreateInfoAMD, VkPipelineCreationFeedbackCreateInfoEXT, VkPipelineDiscardRectangleStateCreateInfoEXT, VkPipelineFragmentShadingRateEnumStateCreateInfoNV, VkPipelineFragmentShadingRateStateCreateInfoKHR, or VkPipelineRepresentativeFragmentTestStateCreateInfoNV

• VUID-VkGraphicsPipelineCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

• VUID-VkGraphicsPipelineCreateInfo-flags-parameter
  flags must be a valid combination of VkPipelineCreateFlagBits values

• VUID-VkGraphicsPipelineCreateInfo-pStages-parameter
  pStages must be a valid pointer to an array of stageCount valid VkPipelineShaderStageCreateInfo structures

• VUID-VkGraphicsPipelineCreateInfo-pRasterizationState-parameter
  pRasterizationState must be a valid pointer to a valid VkPipelineRasterizationStateCreateInfo structure

• VUID-VkGraphicsPipelineCreateInfo-pDynamicState-parameter
  If pDynamicState is not NULL, pDynamicState must be a valid pointer to a valid VkPipelineDynamicStateCreateInfo structure

• VUID-VkGraphicsPipelineCreateInfo-layout-parameter
  layout must be a valid VkPipelineLayout handle

• VUID-VkGraphicsPipelineCreateInfo-renderPass-parameter
  renderPass must be a valid VkRenderPass handle

• VUID-VkGraphicsPipelineCreateInfo-stageCount-arraylength
  stageCount must be greater than 0

• VUID-VkGraphicsPipelineCreateInfo-commonparent
  Each of basePipelineHandle, layout, and renderPass that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

Possible values of the flags member of VkGraphicsPipelineCreateInfo, VkRayTracingPipelineCreateInfoKHR, VkRayTracingPipelineCreateInfoNV, and VkComputePipelineCreateInfo, specifying how a pipeline is created, are:
typedef enum VkPipelineCreateFlagBits {
    VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT = 0x00000001,
    VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT = 0x00000002,
    VK_PIPELINE_CREATE_DERIVATIVE_BIT = 0x00000004,
    // Provided by VK_VERSION_1_1
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT = 0x00000008,
    // Provided by VK_VERSION_1_1
    VK_PIPELINE_CREATE_DISPATCH_BASE_BIT = 0x00000010,
    // Provided by VK_KHR_ray_tracing_pipeline
    VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR = 0x00004000,
    // Provided by VK_KHR_ray_tracing_pipeline
    VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR = 0x00008000,
    // Provided by VK_KHR_ray_tracing_pipeline
    VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR = 0x00010000,
    // Provided by VK_KHR_ray_tracing_pipeline
    VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR = 0x00020000,
    // Provided by VK_KHR_device_group
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT_KHR = 0x00000008,
    // Provided by VK_KHR_device_group
    VK_PIPELINE_CREATE_DISPATCH_BASE_KHR = VK_PIPELINE_CREATE_DISPATCH_BASE_BIT,
    // Provided by VK_KHR_device_group
    VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR = 0x00080000,
    // Provided by VK_NV_ray_tracing
    VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV = 0x00000020,
    // Provided by VK_KHR_pipeline_executable_properties
    VK_PIPELINE_CREATE_CAPTURE_STATISTICS_BIT_KHR = 0x00000040,
    // Provided by VK_KHR_pipeline_executable_properties
    VK_PIPELINE_CREATE_CAPTURE_INTERNAL_REPRESENTATIONS_BIT_KHR = 0x00000080,
    // Provided by VK_NV_device_generated_commands
    VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV = 0x00040000,
    // Provided by VK_KHR_pipeline_library
    VK_PIPELINE_CREATE_LIBRARY_BIT_KHR = 0x00000800,
    // Provided by VK_EXT_pipeline_creation_cache_control
    VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT = 0x00000100,
    // Provided by VK_EXT_pipeline_creation_cache_control
    VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT = 0x00000200,
    VK_PIPELINE_CREATE_DISPATCH_BASE = VK_PIPELINE_CREATE_DISPATCH_BASE_BIT,
    // Provided by VK_KHR_device_group
    VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT_KHR = VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT,
    // Provided by VK_KHR_device_group
    VK_PIPELINE_CREATE_DISPATCH_BASE_KHR = VK_PIPELINE_CREATE_DISPATCH_BASE_KHR,
} VkPipelineCreateFlagBits;

• **VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT** specifies that the created pipeline will not be optimized. Using this flag may reduce the time taken to create the pipeline.

• **VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT** specifies that the pipeline to be created is allowed to
be the parent of a pipeline that will be created in a subsequent pipeline creation call.

- **VK_PIPELINE_CREATE_DERIVATIVE_BIT** specifies that the pipeline to be created will be a child of a previously created parent pipeline.

- **VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT** specifies that any shader input variables decorated as ViewIndex will be assigned values as if they were decorated as DeviceIndex.

- **VK_PIPELINE_CREATE_DISPATCH_BASE** specifies that a compute pipeline can be used with vkCmdDispatchBase with a non-zero base workgroup.

- **VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV** specifies that a pipeline is created with all shaders in the deferred state. Before using the pipeline the application must call vkCompileDeferredNV exactly once on each shader in the pipeline before using the pipeline.

- **VK_PIPELINE_CREATE_CAPTURE_STATISTICS_BIT_KHR** specifies that the shader compiler should capture statistics for the executables produced by the compile process which can later be retrieved by calling vkGetPipelineExecutableStatisticsKHR. Enabling this flag must not affect the final compiled pipeline but may disable pipeline caching or otherwise affect pipeline creation time.

- **VK_PIPELINE_CREATE_CAPTURE_INTERNAL_REPRESENTATIONS_BIT_KHR** specifies that the shader compiler should capture the internal representations of executables produced by the compile process which can later be retrieved by calling vkGetPipelineExecutableInternalRepresentationsKHR. Enabling this flag must not affect the final compiled pipeline but may disable pipeline caching or otherwise affect pipeline creation time.

- **VK_PIPELINE_CREATE_LIBRARY_BIT_KHR** specifies that the pipeline cannot be used directly, and instead defines a pipeline library that can be combined with other pipelines using the VkPipelineLibraryCreateInfoKHR structure. This is available in ray tracing pipelines.

- **VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR** specifies that an any-hit shader will always be present when an any-hit shader would be executed.

- **VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR** specifies that a closest hit shader will always be present when a closest hit shader would be executed.

- **VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR** specifies that a miss shader will always be present when a miss shader would be executed.

- **VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR** specifies that an intersection shader will always be present when an intersection shader would be executed.

- **VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR** specifies that triangle primitives will be skipped during traversal using OpTraceRayKHR.

- **VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR** specifies that AABB primitives will be skipped during traversal using OpTraceRayKHR.

- **VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR** specifies that the shader group handles can be saved and reused on a subsequent run (e.g. for trace capture and replay).

- **VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV** specifies that the pipeline can be used in combination with Device-Generated Commands.

- **VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT** specifies that pipeline creation
will fail if a compile is required for creation of a valid `VkPipeline` object; `VK_PIPELINE_COMPILE_REQUIRED_EXT` will be returned by pipeline creation, and the `VkPipeline` will be set to `VK_NULL_HANDLE`.

- When creating multiple pipelines, `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT` specifies that control will be returned to the application on failure of the corresponding pipeline rather than continuing to create additional pipelines.

It is valid to set both `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` and `VK_PIPELINE_CREATE_DERIVATIVE_BIT`. This allows a pipeline to be both a parent and possibly a child in a pipeline hierarchy. See Pipeline Derivatives for more information.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineCreateFlags;
```

`VkPipelineCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineCreateFlagBits`.

The `VkPipelineDynamicStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType                      sType;  // is the type of this structure.
    const void*                          pNext;  // is NULL or a pointer to a structure extending this structure.
    VkPipelineDynamicStateCreateFlags    flags;  // is reserved for future use.
    uint32_t                             dynamicStateCount;  // is the number of elements in the pDynamicStates array.
    const VkDynamicState*                pDynamicStates;  // is a pointer to an array of VkDynamicState values specifying which pieces of pipeline state will use the values from dynamic state commands rather than from pipeline state creation info.
} VkPipelineDynamicStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `dynamicStateCount` is the number of elements in the `pDynamicStates` array.
- `pDynamicStates` is a pointer to an array of `VkDynamicState` values specifying which pieces of pipeline state will use the values from dynamic state commands rather than from pipeline state creation info.

**Valid Usage**

- `VUID-VkPipelineDynamicStateCreateInfo-pDynamicStates-01442` Each element of `pDynamicStates` must be unique
Valid Usage (Implicit)

- VUID-VkPipelineDynamicStateCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO

- VUID-VkPipelineDynamicStateCreateInfo-pNext-pNext
  pNext must be NULL

- VUID-VkPipelineDynamicStateCreateInfo-flags-zerobitmask
  flags must be 0

- VUID-VkPipelineDynamicStateCreateInfo-pDynamicStates-parameter
  If dynamicStateCount is not 0, pDynamicStates must be a valid pointer to an array of
dynamicStateCount valid VkDynamicState values

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineDynamicStateCreateFlags;

VkPipelineDynamicStateCreateFlags is a bitmask type for setting a mask, but is currently reserved for
future use.

The source of different pieces of dynamic state is specified by the
VkPipelineDynamicStateCreateInfo::pDynamicStates property of the currently active pipeline, each
of whose elements must be one of the values:

// Provided by VK_VERSION_1_0
typedef enum VkDynamicState {
    VK_DYNAMIC_STATE_VIEWPORT = 0,
    VK_DYNAMIC_STATE_SCISSOR = 1,
    VK_DYNAMIC_STATE_LINE_WIDTH = 2,
    VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
    VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
    VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
    VK_DYNAMIC_STATE_STENCILCOMPAREMASK = 6,
    VK_DYNAMIC_STATE_STENCILWRITEMASK = 7,
    VK_DYNAMIC_STATE_STENCILREFERENCE = 8,
    // Provided by VK_NV_clip_space_w_scaling
    VK_DYNAMIC_STATE_VIEWPORTWSCALE_NV = 1000087000,
    // Provided by VK_EXT_discard_rectangles
    VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT = 1000099000,
    // Provided by VK_EXT_sample_locations
    VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT = 1000143000,
    // Provided by VK_KHR_ray_tracing_pipeline
    VK_DYNAMIC_STATE_RAY_TRACING_PIPELINESTACK_SIZE_KHR = 1000347000,
    // Provided by VK_NV_shading_rate_image
    VK_DYNAMIC_STATE_VIEWPORTSHADINGRATERATE_PALETTE_NV = 1000164004,
    // Provided by VK_NV_shading_rate_image
    VK_DYNAMIC_STATE_VIEWPORTCOARSE_SAMPLEORDER_NV = 1000164006,
    // Provided by VK_NV_scissor_exclusive
};
• **VK_DYNAMIC_STATE_VIEWPORT** specifies that the `pViewports` state in `VkPipelineViewportStateCreateInfo` will be ignored and **must** be set dynamically with `vkCmdSetViewport` before any draw commands. The number of viewports used by a pipeline is still specified by the `viewportCount` member of `VkPipelineViewportStateCreateInfo`.

• **VK_DYNAMIC_STATE_SCISSOR** specifies that the `pScissors` state in `VkPipelineViewportStateCreateInfo` will be ignored and must be set dynamically with `vkCmdSetScissor`. The number of viewports used by a pipeline is still specified by the `viewportCount` member of `VkPipelineViewportStateCreateInfo`.
**VkPipelineViewportStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetScissor** before any draw commands. The number of scissor rectangles used by a pipeline is still specified by the **scissorCount** member of **VkPipelineViewportStateCreateInfo**.

- **VK_DYNAMIC_STATE_LINE_WIDTH** specifies that the **lineWidth** state in **VkPipelineRasterizationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetLineWidth** before any draw commands that generate line primitives for the rasterizer.

- **VK_DYNAMIC_STATE_DEPTH_BIAS** specifies that the **depthBiasConstantFactor**, **depthBiasClamp** and **depthBiasSlopeFactor** states in **VkPipelineRasterizationStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthBias** before any draws are performed with **depthBiasEnable** in **VkPipelineRasterizationStateCreateInfo** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_BLEND_CONSTANTS** specifies that the **blendConstants** state in **VkPipelineColorBlendStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetBlendConstants** before any draws are performed with a pipeline state with **VkPipelineColorBlendAttachmentState** member **blendEnable** set to **VK_TRUE** and any of the blend functions using a constant blend color.

- **VK_DYNAMIC_STATE_DEPTH_BOUNDS** specifies that the **minDepthBounds** and **maxDepthBounds** states of **VkPipelineDepthStencilStateCreateInfo** will be ignored and **must** be set dynamically with **vkCmdSetDepthBounds** before any draws are performed with a pipeline state with **VkPipelineDepthStencilStateCreateInfo** member **depthBoundsTestEnable** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK** specifies that the **compareMask** state in **VkPipelineDepthStencilStateCreateInfo** for both **front** and **back** will be ignored and **must** be set dynamically with **vkCmdSetStencilCompareMask** before any draws are performed with a pipeline state with **VkPipelineDepthStencilStateCreateInfo** member **stencilTestEnable** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_STENCIL_WRITE_MASK** specifies that the **writeMask** state in **VkPipelineDepthStencilStateCreateInfo** for both **front** and **back** will be ignored and **must** be set dynamically with **vkCmdSetStencilWriteMask** before any draws are performed with a pipeline state with **VkPipelineDepthStencilStateCreateInfo** member **stencilTestEnable** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_STENCIL_REFERENCE** specifies that the **reference** state in **VkPipelineDepthStencilStateCreateInfo** for both **front** and **back** will be ignored and **must** be set dynamically with **vkCmdSetStencilReference** before any draws are performed with a pipeline state with **VkPipelineDepthStencilStateCreateInfo** member **stencilTestEnable** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV** specifies that the **pViewportScalings** state in **VkPipelineViewportWScalingStateCreateInfoNV** will be ignored and **must** be set dynamically with **vkCmdSetViewportWScalingNV** before any draws are performed with a pipeline state with **VkPipelineViewportWScalingStateCreateInfoNV** member **viewportScalingEnable** set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT** specifies that the **pDiscardRectangles** state in **VkPipelineDiscardRectangleStateCreateInfoEXT** will be ignored and **must** be set dynamically with **vkCmdSetDiscardRectangleEXT** before any draw or clear commands. The **VkDiscardRectangleModeEXT** and the number of active discard rectangles is still specified by the **discardRectangleMode** and **discardRectangleCount** members of **VkPipelineDiscardRectangleStateCreateInfoEXT**.

- **VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT** specifies that the **sampleLocationsInfo** state in
VkPipelineSampleLocationsStateCreateInfoEXT will be ignored and **must** be set dynamically with vkCmdSetSampleLocationsEXT before any draw or clear commands. Enabling custom sample locations is still indicated by the sampleLocationsEnable member of VkPipelineSampleLocationsStateCreateInfoEXT.

- **VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV** specifies that the pExclusiveScissors state in VkPipelineViewportExclusiveScissorStateCreateInfoNV will be ignored and **must** be set dynamically with vkCmdSetExclusiveScissorNV before any draw commands. The number of exclusive scissor rectangles used by a pipeline is still specified by the exclusiveScissorCount member of VkPipelineViewportExclusiveScissorStateCreateInfoNV.

- **VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV** specifies that the pShadingRatePalettes state in VkPipelineViewportShadingRateImageStateCreateInfoNV will be ignored and **must** be set dynamically with vkCmdSetViewportShadingRatePaletteNV before any draw commands.

- **VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV** specifies that the coarse sample order state in VkPipelineViewportCoarseSampleOrderStateCreateInfoNV will be ignored and **must** be set dynamically with vkCmdSetCoarseSampleOrderNV before any draw commands.

- **VK_DYNAMIC_STATE_LINE_STIPPLE_EXT** specifies that the lineStippleFactor and lineStipplePattern state in VkPipelineRasterizationLineStateCreateInfoEXT will be ignored and **must** be set dynamically with vkCmdSetLineStippleEXT before any draws are performed with a pipeline state with VkPipelineRasterizationLineStateCreateInfoEXT member stippledLineEnable set to **VK_TRUE**.

- **VK_DYNAMIC_STATE_CULL_MODE_EXT** specifies that the cullMode state in VkPipelineRasterizationStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetCullModeEXT before any draw commands.

- **VK_DYNAMIC_STATE_FRONT_FACE_EXT** specifies that the frontFace state in VkPipelineRasterizationStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetFrontFaceEXT before any draw commands.

- **VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT** specifies that the topology state in VkPipelineInputAssemblyStateCreateInfo only specifies the topology class, and the specific topology order and adjacency **must** be set dynamically with vkCmdSetPrimitiveTopologyEXT before any draw commands.

- **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** specifies that the viewportCount and pViewports state in VkPipelineViewportStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetViewportWithCountEXT before any draw call.

- **VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT** specifies that the scissorCount and pScissors state in VkPipelineViewportStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetScissorWithCountEXT before any draw call.

- **VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT** specifies that the stride state in VkVertexInputBindingDescription will be ignored and **must** be set dynamically with vkCmdBindVertexBuffers2EXT before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT** specifies that the depthTestEnable state in VkPipelineDepthStencilStateCreateInfo will be ignored and **must** be set dynamically with vkCmdSetDepthTestEnableEXT before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT** specifies that the depthWriteEnable state in
VkPipelineDepthStencilStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetDepthWriteEnableEXT` before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT** specifies that the `depthCompareOp` state in VkPipelineDepthStencilStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetDepthCompareOpEXT` before any draw call.

- **VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT** specifies that the `depthBoundsTestEnable` state in VkPipelineDepthStencilStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetDepthBoundsTestEnableEXT` before any draw call.

- **VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT** specifies that the `stencilTestEnable` state in VkPipelineDepthStencilStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetStencilTestEnableEXT` before any draw call.

- **VK_DYNAMIC_STATE_STENCIL_OP_EXT** specifies that the `failOp`, `passOp`, `depthFailOp`, and `compareOp` states in VkPipelineDepthStencilStateCreateInfo for both front and back will be ignored and **must** be set dynamically with `vkCmdSetStencilOpEXT` before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member `stencilTestEnable` set to **VK_TRUE**

- **VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT** specifies that the `patchControlPoints` state in VkPipelineTessellationStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetPatchControlPointsEXT` before any draw commands.

- **VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT** specifies that the `rasterizerDiscardEnable` state in VkPipelineRasterizationStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetRasterizerDiscardEnableEXT` before any draw commands.

- **VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT** specifies that the `depthBiasEnable` state in VkPipelineRasterizationStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetDepthBiasEnableEXT` before any draw commands.

- **VK_DYNAMIC_STATE_LOGIC_OP_EXT** specifies that the `logicOp` state in VkPipelineColorBlendStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetLogicOpEXT` before any draw commands.

- **VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT** specifies that the `primitiveRestartEnable` state in VkPipelineInputAssemblyStateCreateInfo will be ignored and **must** be set dynamically with `vkCmdSetPrimitiveRestartEnableEXT` before any draw commands.

- **VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR** specifies that state in VkPipelineFragmentShadingRateStateCreateInfoKHR and VkPipelineFragmentShadingRateEnumStateCreateInfoNV will be ignored and **must** be set dynamically with `vkCmdSetFragmentShadingRateKHR` or `vkCmdSetFragmentShadingRateEnumNV` before any draw commands.

- **VK_DYNAMIC_STATE_RAY_TRACING_PIPELINE_STACK_SIZE_KHR** specifies that the default stack size computation for the pipeline will be ignored and **must** be set dynamically with `vkCmdSetRayTracingPipelineStackSizeKHR` before any ray tracing calls are performed.

- **VK_DYNAMIC_STATE_VERTEX_INPUT_EXT** specifies that the `pVertexInputState` states will be ignored and **must** be set dynamically with `vkCmdSetVertexInputEXT` before any draw commands.

- **VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT** specifies that the `pColorWriteEnables` state in VkPipelineColorWriteCreateInfoEXT will be ignored and **must** be set dynamically with
vkCmdSetColorWriteEnableEXT before any draw call.

### 10.2.1. Valid Combinations of Stages for Graphics Pipelines

The geometric primitive processing can either be handled on a per primitive basis by the vertex, tessellation, and geometry shader stages, or on a per mesh basis using task and mesh shader stages. If the pipeline includes a mesh shader stage, it uses the mesh pipeline, otherwise it uses the primitive pipeline.

If a task shader is omitted, the task shading stage is skipped.

If tessellation shader stages are omitted, the tessellation shading and fixed-function stages of the pipeline are skipped.

If a geometry shader is omitted, the geometry shading stage is skipped.

If a fragment shader is omitted, fragment color outputs have undefined values, and the fragment depth value is unmodified. This can be useful for depth-only rendering.

Presence of a shader stage in a pipeline is indicated by including a valid `VkPipelineShaderStageCreateInfo` with `module` and `pName` selecting an entry point from a shader module, where that entry point is valid for the stage specified by `stage`.

Presence of some of the fixed-function stages in the pipeline is implicitly derived from enabled shaders and provided state. For example, the fixed-function tessellator is always present when the pipeline has valid Tessellation Control and Tessellation Evaluation shaders.

*For example:*

- Depth/stencil-only rendering in a subpass with no color attachments
  - Active Pipeline Shader Stages
    - Vertex Shader
  - Required: Fixed-Function Pipeline Stages
    - `VkPipelineVertexInputStateCreateInfo`
    - `VkPipelineInputAssemblyStateCreateInfo`
    - `VkPipelineViewportStateCreateInfo`
    - `VkPipelineRasterizationStateCreateInfo`
    - `VkPipelineMultisampleStateCreateInfo`
    - `VkPipelineDepthStencilStateCreateInfo`

- Color-only rendering in a subpass with no depth/stencil attachment
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Fragment Shader
  - Required: Fixed-Function Pipeline Stages
    - `VkPipelineVertexInputStateCreateInfo`
• Rendering pipeline with tessellation and geometry shaders
  ◦ Active Pipeline Shader Stages
    • Vertex Shader
    • Tessellation Control Shader
    • Tessellation Evaluation Shader
    • Geometry Shader
    • Fragment Shader
  ◦ Required: Fixed-Function Pipeline Stages
    • VkPipelineVertexInputStateCreateInfo
    • VkPipelineInputAssemblyStateCreateInfo
    • VkPipelineTessellationStateCreateInfo
    • VkPipelineViewportStateCreateInfo
    • VkPipelineRasterizationStateCreateInfo
    • VkPipelineMultisampleStateCreateInfo
    • VkPipelineColorBlendStateCreateInfo

• Rendering pipeline with task and mesh shaders
  ◦ Active Pipeline Shader Stages
    • Task Shader
    • Mesh Shader
    • Fragment Shader
  ◦ Required: Fixed-Function Pipeline Stages
    • VkPipelineViewportStateCreateInfo
    • VkPipelineRasterizationStateCreateInfo
    • VkPipelineMultisampleStateCreateInfo
    • VkPipelineDepthStencilStateCreateInfo
    • VkPipelineColorBlendStateCreateInfo

10.2.2. Graphics Pipeline Shader Groups

Graphics pipelines can contain multiple shader groups that can be bound individually. Each shader
group behaves as if it was a pipeline using the shader group’s state. When the pipeline is bound by regular means, it behaves as if the state of group 0 is active, use vkCmdBindPipelineShaderGroupNV to bind an individual shader group.

The primary purpose of shader groups is allowing the device to bind different pipeline state using Device-Generated Commands. The VkGraphicsPipelineShaderGroupsCreateInfoNV structure is defined as:

```c
// Provided by VK_NV_device_generated_commands
typedef struct VkGraphicsPipelineShaderGroupsCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    uint32_t groupCount;
    const VkGraphicsShaderGroupCreateInfoNV* pGroups;
    uint32_t pipelineCount;
    const VkPipeline* pPipelines;
} VkGraphicsPipelineShaderGroupsCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `groupCount` is the number of elements in the `pGroups` array.
- `pGroups` is an array of VkGraphicsShaderGroupCreateInfoNV values specifying which state of the original VkGraphicsPipelineCreateInfo each shader group overrides.
- `pipelineCount` is the number of elements in the `pPipelines` array.
- `pPipelines` is an array of graphics VkPipeline, which are referenced within the created pipeline, including all their shader groups.

When referencing shader groups by index, groups defined in the referenced pipelines are treated as if they were defined as additional entries in `pGroups`. They are appended in the order they appear in the `pPipelines` array and in the `pGroups` array when those pipelines were defined.

The application must maintain the lifetime of all such referenced pipelines based on the pipelines that make use of them.
Valid Usage

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-groupCount-02879
  
  \textit{groupCount} \textbf{must} \textbf{be at least} \textbf{1} \textbf{and as maximum} \textbf{VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxGraphicsShaderGroupCount}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-groupCount-02880
  
  The sum of \textit{groupCount} including those groups added from referenced \textit{pPipelines} \textbf{must also be as maximum} \textbf{VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxGraphicsShaderGroupCount}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pGroups-02881
  
  The state of the first element of \textit{pGroups} \textbf{must match its equivalent within the parent’s VkGraphicsPipelineCreateInfo}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pGroups-02882
  
  Each element of \textit{pGroups} \textbf{must in combination with the rest of the pipeline state yield a valid state configuration}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pGroups-02884
  
  All elements of \textit{pGroups} \textbf{must use the same shader stage combinations unless any mesh shader stage is used, then either combination of task and mesh or just mesh shader is valid}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pGroups-02885
  
  Mesh and regular primitive shading stages cannot be mixed across \textit{pGroups}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pPipelines-02886
  
  Each element of \textit{pPipelines} \textbf{must have been created with identical state to the pipeline currently created except the state that can be overridden by VkGraphicsShaderGroupCreateInfoNV}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-deviceGeneratedCommands-02887
  
  The \textbf{VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands feature must be enabled}

Valid Usage (Implicit)

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-sType-sType
  
  \textit{sType} \textbf{must be} \textbf{VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_SHADER_GROUPS_CREATE_INFO_NV}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pGroups-parameter
  
  \textit{pGroups} \textbf{must be a valid pointer to an array of} \textbf{groupCount} \textbf{valid VkGraphicsShaderGroupCreateInfoNV structures}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-pPipelines-parameter
  
  If \textit{pipelineCount} is not 0, \textit{pPipelines} \textbf{must be a valid pointer to an array of} \textbf{pipelineCount} \textbf{valid VkPipeline handles}

- VUID-VkGraphicsPipelineShaderGroupsCreateInfoNV-groupCount-arraylength
  
  \textit{groupCount} \textbf{must be greater than 0}
The `VkGraphicsShaderGroupCreateInfoNV` structure provides the state overrides for each shader group. Each shader group behaves like a pipeline that was created from its state as well as the remaining parent’s state. It is defined as:

```c
// Provided by VK_NV_device_generated_commands
typedef struct VkGraphicsShaderGroupCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo* pStages;
    const VkPipelineVertexInputStateCreateInfo* pVertexInputState;
    const VkPipelineTessellationStateCreateInfo* pTessellationState;
} VkGraphicsShaderGroupCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `stageCount` is the number of entries in the `pStages` array.
- `pStages` is an array of size `stageCount` structures of type `VkPipelineShaderStageCreateInfo` describing the set of the shader stages to be included in this shader group.
- `pVertexInputState` is a pointer to an instance of the `VkPipelineVertexInputStateCreateInfo` structure.
- `pTessellationState` is a pointer to an instance of the `VkPipelineTessellationStateCreateInfo` structure, and is ignored if the shader group does not include a tessellation control shader stage and tessellation evaluation shader stage.

**Valid Usage**

- `VUID-VkGraphicsShaderGroupCreateInfoNV-stageCount-02888`
  - For `stageCount`, the same restrictions as in `VkGraphicsPipelineCreateInfo::stageCount` apply

- `VUID-VkGraphicsShaderGroupCreateInfoNV-pStages-02889`
  - For `pStages`, the same restrictions as in `VkGraphicsPipelineCreateInfo::pStages` apply

- `VUID-VkGraphicsShaderGroupCreateInfoNV-pVertexInputState-02890`
  - For `pVertexInputState`, the same restrictions as in `VkGraphicsPipelineCreateInfo::pVertexInputState` apply

- `VUID-VkGraphicsShaderGroupCreateInfoNV-pTessellationState-02891`
  - For `pTessellationState`, the same restrictions as in `VkGraphicsPipelineCreateInfo::pTessellationState` apply
Valid Usage (Implicit)

- VUID-VkGraphicsShaderGroupCreateInfoNV-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_GRAPHICS_SHADER_GROUP_CREATE_INFO_NV**

- VUID-VkGraphicsShaderGroupCreateInfoNV-pNext-pNext
  
  **pNext** must be **NULL**

- VUID-VkGraphicsShaderGroupCreateInfoNV-pStages-parameter
  
  **pStages** must be a valid pointer to an array of **stageCount** valid **VkPipelineShaderStageCreateInfo** structures

- VUID-VkGraphicsShaderGroupCreateInfoNV-stageCount-arraylength
  
  **stageCount** must be greater than **0**

10.3. Ray Tracing Pipelines

Ray tracing pipelines consist of multiple shader stages, fixed-function traversal stages, and a pipeline layout.

To create ray tracing pipelines, call:

```c
// Provided by VK_NV_ray_tracing
VkResult vkCreateRayTracingPipelinesNV(
    VkDevice                                    device, 
    VkPipelineCache                             pipelineCache, 
    uint32_t                                    createInfoCount, 
    const                                        VkRayTracingPipelineCreateInfoNV*     pCreateInfos, 
    const                                        VkAllocationCallbacks*                pAllocator, 
    VkPipeline*                                 pPipelines); 
```

- **device** is the logical device that creates the ray tracing pipelines.

- **pipelineCache** is either **VK_NULL_HANDLE**, indicating that pipeline caching is disabled, or the handle of a valid pipeline cache object, in which case use of that cache is enabled for the duration of the command.

- **createInfoCount** is the length of the **pCreateInfos** and **pPipelines** arrays.

- **pCreateInfos** is a pointer to an array of **VkRayTracingPipelineCreateInfoNV** structures.

- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

- **pPipelines** is a pointer to an array in which the resulting ray tracing pipeline objects are returned.
Valid Usage

• VUID-vkCreateRayTracingPipelinesNV-flags-03415
  If the flags member of any element of pCreateInfos contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and the basePipelineIndex member of that same element is not -1, basePipelineIndex must be less than the index into pCreateInfos that corresponds to that element

• VUID-vkCreateRayTracingPipelinesNV-flags-03416
  If the flags member of any element of pCreateInfos contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, the base pipeline must have been created with the VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT flag set

• VUID-vkCreateRayTracingPipelinesNV-flags-03816
  flags must not contain the VK_PIPELINE_CREATE_DISPATCH_BASE flag

• VUID-vkCreateRayTracingPipelinesNV-pipelineCache-02903
  If pipelineCache was created with VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT, host access to pipelineCache must be externally synchronized

Valid Usage (Implicit)

• VUID-vkCreateRayTracingPipelinesNV-device-parameter
device must be a valid VkDevice handle

• VUID-vkCreateRayTracingPipelinesNV-pipelineCache-parameter
If pipelineCache is not VK_NULL_HANDLE, pipelineCache must be a valid VkPipelineCache handle

• VUID-vkCreateRayTracingPipelinesNV-pCreateInfos-parameter
pCreateInfos must be a valid pointer to an array of createInfoCount valid VkRayTracingPipelineCreateInfoNV structures

• VUID-vkCreateRayTracingPipelinesNV-pAllocator-parameter
If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• VUID-vkCreateRayTracingPipelinesNV-pPipelines-parameter
pPipelines must be a valid pointer to an array of createInfoCount VkPipeline handles

• VUID-vkCreateRayTracingPipelinesNV-createInfoCount-arraylength
createInfoCount must be greater than 0

• VUID-vkCreateRayTracingPipelinesNV-pipelineCache-parent
If pipelineCache is a valid handle, it must have been created, allocated, or retrieved from device
### Return Codes

**Success**
- `VK_SUCCESS`
- `VK_PIPELINE_COMPILE_REQUIRED_EXT`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_SHADER_NV`

To create ray tracing pipelines, call:

```c
// Provided by VK_KHR_ray_tracing_pipeline
VkResult vkCreateRayTracingPipelinesKHR(
    VkDevice                                    device,
    VkDeferredOperationKHR                      deferredOperation,
    VkPipelineCache                             pipelineCache,
    uint32_t                                    createInfoCount,
    const VkRayTracingPipelineCreateInfoKHR*    pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkPipeline*                                 pPipelines);
```

- `device` is the logical device that creates the ray tracing pipelines.
- `deferredOperation` is an optional `VkDeferredOperationKHR` to request deferral for this command.
- `pipelineCache` is either `VK_NULL_HANDLE`, indicating that pipeline caching is disabled, or the handle of a valid `pipeline cache` object, in which case use of that cache is enabled for the duration of the command.
- `createInfoCount` is the length of the `pCreateInfos` and `pPipelines` arrays.
- `pCreateInfos` is a pointer to an array of `VkRayTracingPipelineCreateInfoKHR` structures.
- `pAllocator` controls host memory allocation as described in the `Memory Allocation` chapter.
- `pPipelines` is a pointer to an array in which the resulting ray tracing pipeline objects are returned.

The `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS` error is returned if the implementation is unable to re-use the shader group handles provided in `VkRayTracingShaderGroupCreateInfoKHR` ::`pShaderGroupCaptureReplayHandle` when `VkPhysicalDeviceRayTracingPipelineFeaturesKHR` ::`rayTracingPipelineShaderGroupHandleCaptureReplay` is enabled.
Valid Usage

- **VUID-vkCreateRayTracingPipelinesKHR-flags-03415**
  If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and the `basePipelineIndex` member of that same element is not `-1`, `basePipelineIndex` must be less than the index into `pCreateInfos` that corresponds to that element.

- **VUID-vkCreateRayTracingPipelinesKHR-flags-03416**
  If the `flags` member of any element of `pCreateInfos` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, the base pipeline must have been created with the `VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT` flag set.

- **VUID-vkCreateRayTracingPipelinesKHR-flags-03816**
  `flags` must not contain the `VK_PIPELINE_CREATE_DISPATCH_BASE` flag.

- **VUID-vkCreateRayTracingPipelinesKHR-pipelineCache-02903**
  If `pipelineCache` was created with `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT`, host access to `pipelineCache` must be externally synchronized.

- **VUID-vkCreateRayTracingPipelinesKHR-deferredOperation-03677**
  If `deferredOperation` is not `VK_NULL_HANDLE`, it must be a valid `VkDeferredOperationKHR` object.

- **VUID-vkCreateRayTracingPipelinesKHR-deferredOperation-03678**
  Any previous deferred operation that was associated with `deferredOperation` must be complete.

- **VUID-vkCreateRayTracingPipelinesKHR-rayTracingPipeline-03586**
  The `rayTracingPipeline` feature must be enabled.

- **VUID-vkCreateRayTracingPipelinesKHR-deferredOperation-03587**
  If `deferredOperation` is not `VK_NULL_HANDLE`, the `flags` member of elements of `pCreateInfos` must not include `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT`.
Valid Usage (Implicit)

- **VUID-vkCreateRayTracingPipelinesKHR-device-parameter**
  
  device must be a valid VkDevice handle

- **VUID-vkCreateRayTracingPipelinesKHR-deferredOperation-parameter**
  
  If deferredOperation is not VK_NULL_HANDLE, deferredOperation must be a valid VkDeferredOperationKHR handle

- **VUID-vkCreateRayTracingPipelinesKHR-pipelineCache-parameter**
  
  If pipelineCache is not VK_NULL_HANDLE, pipelineCache must be a valid VkPipelineCache handle

- **VUID-vkCreateRayTracingPipelinesKHR-pCreateInfos-parameter**
  
  pCreateInfos must be a valid pointer to an array of createInfoCount valid VkRayTracingPipelineCreateInfoKHR structures

- **VUID-vkCreateRayTracingPipelinesKHR-pAllocator-parameter**
  
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- **VUID-vkCreateRayTracingPipelinesKHR-pPipelines-parameter**
  
  pPipelines must be a valid pointer to an array of createInfoCount VkPipeline handles

- **VUID-vkCreateRayTracingPipelinesKHR-createInfoCount-arraylength**
  
  createInfoCount must be greater than 0

- **VUID-vkCreateRayTracingPipelinesKHR-deferredOperation-parent**
  
  If deferredOperation is a valid handle, it must have been created, allocated, or retrieved from device

- **VUID-vkCreateRayTracingPipelinesKHR-pipelineCache-parent**
  
  If pipelineCache is a valid handle, it must have been created, allocated, or retrieved from device

Return Codes

Success

- VK_SUCCESS
- VK_OPERATION_DEFERRED_KHR
- VK_OPERATION_NOT_DEFERRED_KHR
- VK_PIPELINE_COMPILE_REQUIRED_EXT

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS

The VkRayTracingPipelineCreateInfoNV structure is defined as:
typedef struct VkRayTracingPipelineCreateInfoNV {
    VkStructureType                               sType;
    const void*                                   pNext;
    VkPipelineCreateFlags                         flags;
    uint32_t                                      stageCount;
    const VkPipelineShaderStageCreateInfo*        pStages;
    uint32_t                                      groupCount;
    const VkRayTracingShaderGroupCreateInfoNV*    pGroups;
    uint32_t                                      maxRecursionDepth;
    VkPipelineLayout                              layout;
    VkPipeline                                    basePipelineHandle;
    int32_t                                       basePipelineIndex;
} VkRayTracingPipelineCreateInfoNV;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkPipelineCreateFlagBits** specifying how the pipeline will be generated.
- **stageCount** is the number of entries in the **pStages** array.
- **pStages** is an array of size **stageCount** structures of type **VkPipelineShaderStageCreateInfo** describing the set of the shader stages to be included in the ray tracing pipeline.
- **groupCount** is the number of entries in the **pGroups** array.
- **pGroups** is an array of size **groupCount** structures of type **VkRayTracingShaderGroupCreateInfoNV** describing the set of the shader stages to be included in each shader group in the ray tracing pipeline.
- **maxRecursionDepth** is the maximum recursion depth of shaders executed by this pipeline.
- **layout** is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- **basePipelineHandle** is a pipeline to derive from.
- **basePipelineIndex** is an index into the **pCreateInfos** parameter to use as a pipeline to derive from.

The parameters **basePipelineHandle** and **basePipelineIndex** are described in more detail in **Pipeline Derivatives**.
Valid Usage

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03421
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` must be a valid handle to a ray tracing `VkPipeline`.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03422
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` must be a valid index into the calling command's `pCreateInfos` parameter.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03423
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not -1, `basePipelineHandle` must be `VK_NULL_HANDLE`.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03424
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` must be -1.

- VUID-VkRayTracingPipelineCreateInfoNV-pStages-03426
  The shader code for the entry points identified by `pStages`, and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Shader Interfaces chapter.

- VUID-VkRayTracingPipelineCreateInfoNV-layout-03427
  `layout` must be consistent with all shaders specified in `pStages`.

- VUID-VkRayTracingPipelineCreateInfoNV-layout-03428
  The number of resources in `layout` accessible to each shader stage that is used by the pipeline must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03429
  `flags` must not include `VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV`.

- VUID-VkRayTracingPipelineCreateInfoNV-pipelineCreationCacheControl-03424
  If the `pipelineCreationCacheControl` feature is not enabled, `flags` must not include `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT` or `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT`.

- VUID-VkRayTracingPipelineCreateInfoNV-stage-03425
  The `stage` member of at least one element of `pStages` must be `VK_SHADER_STAGE_RAYGEN_KHR`.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03426
  `flags` must not include `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR`.

- VUID-VkRayTracingPipelineCreateInfoNV-maxRecursionDepth-03427
  `maxRecursionDepth` must be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxRecursionDepth`.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03428
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR`.

- VUID-VkRayTracingPipelineCreateInfoNV-flags-03429
  `flags` must not include `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR`.

Chapter 10. Pipelines | 573
- **VUID-VkRayTracingPipelineCreateInfoNV-flags-03460**
  flags must not include `VK_PIPELINE_CREATERAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoNV-flags-03461**
  flags must not include `VK_PIPELINE_CREATERAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoNV-flags-03462**
  flags must not include `VK_PIPELINE_CREATERAY_TRACING_SKIP_AABBSBIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoNV-flags-03463**
  flags must not include `VK_PIPELINE_CREATERAY_TRACING_SKIP_TRIANGLES_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoNV-flags-03588**
  flags must not include `VK_PIPELINE_CREATERAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoNV-flags-02957**
  flags must not include both `VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV` and `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT` at the same time
Valid Usage (Implicit)

- **VUID-VkRayTracingPipelineCreateInfoNV-sType-sType**
  
  \textbf{sType must be} VK\_STRUCTURE\_TYPE\_RAY\_TRACING\_PIPELINE\_CREATE\_INFO\_NV

- **VUID-VkRayTracingPipelineCreateInfoNV-pNext-pNext**
  
  \textbf{pNext must be} NULL or a pointer to a valid instance of VkPipelineCreationFeedbackCreateInfoEXT

- **VUID-VkRayTracingPipelineCreateInfoNV-sType-unique**
  
  The \textbf{sType} value of each struct in the \textbf{pNext} chain \textbf{must be unique}

- **VUID-VkRayTracingPipelineCreateInfoNV-flags-parameter**
  
  \textbf{flags must be} a valid combination of VkPipelineCreateFlagBits values

- **VUID-VkRayTracingPipelineCreateInfoNV-pStages-parameter**
  
  \textbf{pStages must be} a valid pointer to an array of \textbf{stageCount} valid VkPipelineShaderStageCreateInfo structures

- **VUID-VkRayTracingPipelineCreateInfoNV-pGroups-parameter**
  
  \textbf{pGroups must be} a valid pointer to an array of \textbf{groupCount} valid VkRayTracingShaderGroupCreateInfoNV structures

- **VUID-VkRayTracingPipelineCreateInfoNV-layout-parameter**
  
  \textbf{layout must be} a valid VkPipelineLayout handle

- **VUID-VkRayTracingPipelineCreateInfoNV-stageCount-arraylength**
  
  \textbf{stageCount must be} greater than 0

- **VUID-VkRayTracingPipelineCreateInfoNV-groupCount-arraylength**
  
  \textbf{groupCount must be} greater than 0

- **VUID-VkRayTracingPipelineCreateInfoNV-commonparent**
  
  Both of basePipelineHandle, and layout that are valid handles of non-ignored parameters \textbf{must} have been created, allocated, or retrieved from the same VkDevice

The VkRayTracingPipelineCreateInfoKHR structure is defined as:
typedef struct VkRayTracingPipelineCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCreateFlags flags;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo* pStages;
    uint32_t groupCount;
    const VkRayTracingShaderGroupCreateInfoKHR* pGroups;
    uint32_t maxPipelineRayRecursionDepth;
    const VkPipelineLibraryCreateInfoKHR* pLibraryInfo;
    const VkRayTracingPipelineInterfaceCreateInfoKHR* pLibraryInterface;
    const VkPipelineDynamicStateCreateInfo* pDynamicState;
    VkPipelineLayout layout;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
} VkRayTracingPipelineCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is a bitmask of **VkPipelineCreateFlagBits** specifying how the pipeline will be generated.
- **stageCount** is the number of entries in the **pStages** array.
- **pStages** is a pointer to an array of stageCount **VkPipelineShaderStageCreateInfo** structures describing the set of the shader stages to be included in the ray tracing pipeline.
- **groupCount** is the number of entries in the **pGroups** array.
- **pGroups** is a pointer to an array of groupCount **VkRayTracingShaderGroupCreateInfoKHR** structures describing the set of the shader stages to be included in each shader group in the ray tracing pipeline.
- **maxPipelineRayRecursionDepth** is the maximum recursion depth of shaders executed by this pipeline.
- **pLibraryInfo** is a pointer to a **VkPipelineLibraryCreateInfoKHR** structure defining pipeline libraries to include.
- **pLibraryInterface** is a pointer to a **VkRayTracingPipelineInterfaceCreateInfoKHR** structure defining additional information when using pipeline libraries.
- **pDynamicState** is a pointer to a **VkPipelineDynamicStateCreateInfo** structure, and is used to indicate which properties of the pipeline state object are dynamic and can be changed independently of the pipeline state. This can be **NULL**, which means no state in the pipeline is considered dynamic.
- **layout** is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- **basePipelineHandle** is a pipeline to derive from.
- **basePipelineIndex** is an index into the **pCreateInfos** parameter to use as a pipeline to derive from.
The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in Pipeline Derivatives.

When `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR` is specified, this pipeline defines a pipeline library which **cannot** be bound as a ray tracing pipeline directly. Instead, pipeline libraries define common shaders and shader groups which **can** be included in future pipeline creation.

If pipeline libraries are included in `pLibraryInfo`, shaders defined in those libraries are treated as if they were defined as additional entries in `pStages`, appended in the order they appear in the `pLibraries` array and in the `pStages` array when those libraries were defined.

When referencing shader groups in order to obtain a shader group handle, groups defined in those libraries are treated as if they were defined as additional entries in `pGroups`, appended in the order they appear in the `pLibraries` array and in the `pGroups` array when those libraries were defined. The shaders these groups reference are set when the pipeline library is created, referencing those specified in the pipeline library, not in the pipeline that includes it.

The default stack size for a pipeline if `VK_DYNAMIC_STATE_RAY_TRACING_PIPELINE_STACK_SIZE_KHR` is not provided is computed as described in Ray Tracing Pipeline Stack.
Valid Usage

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03421**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is `-1`, `basePipelineHandle` must be a valid handle to a ray tracing `VkPipeline`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03422**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is `VK_NULL_HANDLE`, `basePipelineIndex` must be a valid index into the calling command’s `pCreateInfos` parameter.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03423**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not `-1`, `basePipelineHandle` must be `VK_NULL_HANDLE`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03424**
  If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` must be `-1`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pStages-03426**
  The shader code for the entry points identified by `pStages`, and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Shader Interfaces chapter.

- **VUID-VkRayTracingPipelineCreateInfoKHR-layout-03427**
  `layout` must be consistent with all shaders specified in `pStages`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-layout-03428**
  The number of resources in `layout` accessible to each shader stage that is used by the pipeline must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-02904**
  `flags` must not include `VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pipelineCreationCacheControl-02905**
  If the `pipelineCreationCacheControl` feature is not enabled, `flags` must not include `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT` or `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-stage-03425**
  If `flags` does not include `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR`, the `stage` member of at least one element of `pStages`, including those implicitly added by `pLibraryInfo`, must be `VK_SHADER_STAGE_RAYGEN_BIT_KHR`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-maxPipelineRayRecursionDepth-03589**
  `maxPipelineRayRecursionDepth` must be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxRayRecursionDepth`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03465**
  If `flags` includes `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR`, `pLibraryInterface` must not be `NULL`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-03590**
  If `pLibraryInfo` is not `NULL` and its `libraryCount` member is greater than `0`, its `pLibraryInterface` member must not be `NULL`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraries-03591**
Each element of the `pLibraries` member of `pLibraryInfo` must have been created with the value of `maxPipelineRayRecursionDepth` equal to that in this pipeline.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-03592**
  If `pLibraryInfo` is not `NULL`, each element of its `pLibraries` member must have been created with a layout that is compatible with the layout in this pipeline.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-03593**
  If `pLibraryInfo` is not `NULL`, each element of its `pLibraries` member must have been created with values of the `maxPipelineRayPayloadSize` and `maxPipelineRayHitAttributeSize` members of `pLibraryInterface` equal to those in this pipeline.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03594**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-04718**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-04719**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-04720**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-04721**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-04722**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-04723**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR`, each element of the `pLibraries` member of `libraries` must have been created with the `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR` bit set.

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-03595**
  If the `VK_KHR_pipeline_library` extension is not enabled, `pLibraryInfo` and `pLibraryInterface` must be `NULL`.

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03470**
  If `flags` includes `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR`, for
any element of `pGroups` with a type of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`, the anyHitShader of that element must not be `VK_SHADER_UNUSED_KHR`

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03471**
  If flags includes `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR`, for any element of `pGroups` with a type of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`, the closestHitShader of that element must not be `VK_SHADER_UNUSED_KHR`

- **VUID-VkRayTracingPipelineCreateInfoKHR-rayTraversalPrimitiveCulling-03596**
  If the rayTraversalPrimitiveCulling feature is not enabled, flags must not include `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoKHR-rayTraversalPrimitiveCulling-03597**
  If the rayTraversalPrimitiveCulling feature is not enabled, flags must not include `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-03598**
  If flags includes `VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR`, rayTracingPipelineShaderGroupHandleCaptureReplay must be enabled

- **VUID-VkRayTracingPipelineCreateInfoKHR-rayTracingPipelineShaderGroupHandleCaptureReplay-03599**
  If `VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipelineShaderGroupHandleCaptureReplay` is `VK_TRUE` and the pShaderGroupCaptureReplayHandle member of any element of `pGroups` is not NULL, flags must include `VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR`

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-03600**
  If `pLibraryInfo` is not NULL and its libraryCount is 0, stageCount must not be 0

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-03601**
  If `pLibraryInfo` is not NULL and its libraryCount is 0, groupCount must not be 0

- **VUID-VkRayTracingPipelineCreateInfoKHR-pDynamicStates-03602**
  Any element of the pDynamicStates member of `pDynamicState` must be `VK_DYNAMIC_STATE_RAY_TRACING_PIPELINE_STACK_SIZE_KHR`
Valid Usage (Implicit)

- **VUID-VkRayTracingPipelineCreateInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR`

- **VUID-VkRayTracingPipelineCreateInfoKHR-pNext-pNext**
  
  `pNext` must be `NULL` or a pointer to a valid instance of `VkPipelineCreationFeedbackCreateInfoEXT`

- **VUID-VkRayTracingPipelineCreateInfoKHR-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkRayTracingPipelineCreateInfoKHR-flags-parameter**
  
  `flags` must be a valid combination of `VkPipelineCreateFlagBits` values

- **VUID-VkRayTracingPipelineCreateInfoKHR-pStages-parameter**
  
  If `stageCount` is not 0, `pStages` must be a valid pointer to an array of `stageCount` valid `VkPipelineShaderStageCreateInfo` structures

- **VUID-VkRayTracingPipelineCreateInfoKHR-pGroups-parameter**
  
  If `groupCount` is not 0, `pGroups` must be a valid pointer to an array of `groupCount` valid `VkRayTracingShaderGroupCreateInfoKHR` structures

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInfo-parameter**
  
  If `pLibraryInfo` is not `NULL`, `pLibraryInfo` must be a valid pointer to a valid `VkPipelineLibraryCreateInfoKHR` structure

- **VUID-VkRayTracingPipelineCreateInfoKHR-pLibraryInterface-parameter**
  
  If `pLibraryInterface` is not `NULL`, `pLibraryInterface` must be a valid pointer to a valid `VkRayTracingPipelineInterfaceCreateInfoKHR` structure

- **VUID-VkRayTracingPipelineCreateInfoKHR-pDynamicState-parameter**
  
  If `pDynamicState` is not `NULL`, `pDynamicState` must be a valid pointer to a valid `VkPipelineDynamicStateCreateInfo` structure

- **VUID-VkRayTracingPipelineCreateInfoKHR-layout-parameter**
  
  `layout` must be a valid `VkPipelineLayout` handle

- **VUID-VkRayTracingPipelineCreateInfoKHR-commonparent**
  
  Both of `basePipelineHandle`, and `layout` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

The `VkRayTracingShaderGroupCreateInfoNV` structure is defined as:
// Provided by VK_NV_ray_tracing

typedef struct VkRayTracingShaderGroupCreateInfoNV {
    VkStructureType                   sType;
    const void*                       pNext;
    VkRayTracingShaderGroupTypeKHR    type;
    uint32_t                          generalShader;
    uint32_t                          closestHitShader;
    uint32_t                          anyHitShader;
    uint32_t                          intersectionShader;
} VkRayTracingShaderGroupCreateInfoNV;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **type** is the type of hit group specified in this structure.
- **generalShader** is the index of the ray generation, miss, or callable shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV`, and `VK_SHADER_UNUSED_NV` otherwise.
- **closestHitShader** is the optional index of the closest hit shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_NV`, and `VK_SHADER_UNUSED_NV` otherwise.
- **anyHitShader** is the optional index of the any-hit shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_NV`, and `VK_SHADER_UNUSED_NV` otherwise.
- **intersectionShader** is the index of the intersection shader from `VkRayTracingPipelineCreateInfoNV::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_NV`, and `VK_SHADER_UNUSED_NV` otherwise.
Valid Usage

- VUID-VkRayTracingShaderGroupCreateInfoNV-type-02413
  If type is VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV then generalShader must be a valid index into VkRayTracingPipelineCreateInfoNV::pStages referring to a shader of VK_SHADER_STAGE_RAYGEN_BIT_NV, VK_SHADER_STAGE_MISS_BIT_NV, or VK_SHADER_STAGE_CALLABLE_BIT_NV

- VUID-VkRayTracingShaderGroupCreateInfoNV-type-02414
  If type is VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV then closestHitShader, anyHitShader, and intersectionShader must be VK_SHADER_UNUSED_NV

- VUID-VkRayTracingShaderGroupCreateInfoNV-type-02415
  If type is VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_NV then intersectionShader must be a valid index into VkRayTracingPipelineCreateInfoNV::pStages referring to a shader of VK_SHADER_STAGE_INTERSECTION_BIT_NV

- VUID-VkRayTracingShaderGroupCreateInfoNV-type-02416
  If type is VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV then intersectionShader must be VK_SHADER_UNUSED_NV

- VUID-VkRayTracingShaderGroupCreateInfoNV-closestHitShader-02417
  closestHitShader must be either VK_SHADER_UNUSED_NV or a valid index into VkRayTracingPipelineCreateInfoNV::pStages referring to a shader of VK_SHADER_STAGE_CLOSEST_HIT_BIT_NV

- VUID-VkRayTracingShaderGroupCreateInfoNV-anyHitShader-02418
  anyHitShader must be either VK_SHADER_UNUSED_NV or a valid index into VkRayTracingPipelineCreateInfoNV::pStages referring to a shader of VK_SHADER_STAGE_ANY_HIT_BIT_NV

Valid Usage (Implicit)

- VUID-VkRayTracingShaderGroupCreateInfoNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_NV

- VUID-VkRayTracingShaderGroupCreateInfoNV-pNext-pNext
  pNext must be NULL

- VUID-VkRayTracingShaderGroupCreateInfoNV-type-parameter
  type must be a valid VkRayTracingShaderGroupTypeKHR value

The VkRayTracingShaderGroupCreateInfoKHR structure is defined as:
typedef struct VkRayTracingShaderGroupCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkRayTracingShaderGroupTypeKHR type;
    uint32_t generalShader;
    uint32_t closestHitShader;
    uint32_t anyHitShader;
    uint32_t intersectionShader;
    const void* pShaderGroupCaptureReplayHandle;
} VkRayTracingShaderGroupCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **type** is the type of hit group specified in this structure.
- **generalShader** is the index of the ray generation, miss, or callable shader from `VkRayTracingPipelineCreateInfoKHR::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR`, and `VK_SHADER_UNUSED_KHR` otherwise.
- **closestHitShader** is the optional index of the closest hit shader from `VkRayTracingPipelineCreateInfoKHR::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`, and `VK_SHADER_UNUSED_KHR` otherwise.
- **anyHitShader** is the optional index of the any-hit shader from `VkRayTracingPipelineCreateInfoKHR::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR` or `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`, and `VK_SHADER_UNUSED_KHR` otherwise.
- **intersectionShader** is the index of the intersection shader from `VkRayTracingPipelineCreateInfoKHR::pStages` in the group if the shader group has `type` of `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`, and `VK_SHADER_UNUSED_KHR` otherwise.
- **pShaderGroupCaptureReplayHandle** is an optional pointer to replay information for this shader group. Ignored if `VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipelineShaderGroupHandleCaptureReplay` is `VK_FALSE`. 
Valid Usage

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-type-03474**
  If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR` then `generalShader` must be a valid index into `VkRayTracingPipelineCreateInfoKHR::pStages` referring to a shader of `VK_SHADER_STAGE_RAYGEN_BIT_KHR`, `VK_SHADER_STAGE_MISS_BIT_KHR`, or `VK_SHADER_STAGE_CALLABLE_BIT_KHR`.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-type-03475**
  If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR` then `closestHitShader`, `anyHitShader`, and `intersectionShader` must be `VK_SHADER_UNUSED_KHR`.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-type-03476**
  If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR` then `intersectionShader` must be a valid index into `VkRayTracingPipelineCreateInfoKHR::pStages` referring to a shader of `VK_SHADER_STAGE_INTERSECTION_BIT_KHR`.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-type-03477**
  If `type` is `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR` then `intersectionShader` must be `VK_SHADER_UNUSED_KHR`.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-closestHitShader-03478**
  `closestHitShader` must be either `VK_SHADER_UNUSED_KHR` or a valid index into `VkRayTracingPipelineCreateInfoKHR::pStages` referring to a shader of `VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR`.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-anyHitShader-03479**
  `anyHitShader` must be either `VK_SHADER_UNUSED_KHR` or a valid index into `VkRayTracingPipelineCreateInfoKHR::pStages` referring to a shader of `VK_SHADER_STAGE_ANY_HIT_BIT_KHR`.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-rayTracingPipelineShaderGroupHandleCaptureReplayMixed-03603**
  If `VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipelineShaderGroupHandleCaptureReplayMixed` is `VK_FALSE` then `pShaderGroupCaptureReplayHandle` must not be provided if it has not been provided on a previous call to ray tracing pipeline creation.

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-rayTracingPipelineShaderGroupHandleCaptureReplayMixed-03604**
  If `VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipelineShaderGroupHandleCaptureReplayMixed` is `VK_FALSE` then the caller must guarantee that no ray tracing pipeline creation commands with `pShaderGroupCaptureReplayHandle` provided execute simultaneously with ray tracing pipeline creation commands without `pShaderGroupCaptureReplayHandle` provided.
Valid Usage (Implicit)

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR`

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-pNext-pNext**
  
  `pNext` must be `NULL`

- **VUID-VkRayTracingShaderGroupCreateInfoKHR-type-parameter**
  
  `type` must be a valid `VkRayTracingShaderGroupTypeKHR` value

Possible values of `type` in `VkRayTracingShaderGroupCreateInfoKHR` are:

```c
// Provided by VK_KHR_ray_tracing_pipeline
typedef enum VkRayTracingShaderGroupTypeKHR {
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR = 0,
    VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR = 1,
    VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR = 2,
    // Provided by VK_NV_ray_tracing
    VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_NV =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_NV =
    VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_NV =
} VkRayTracingShaderGroupTypeKHR;
```

or the equivalent

```c
// Provided by VK_NV_ray_tracing
typedef VkRayTracingShaderGroupTypeKHR VkRayTracingShaderGroupTypeNV;
```

- **VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR** indicates a shader group with a single `VK_SHADER_STAGE_RAYGEN_BIT_KHR`, `VK_SHADER_STAGE_MISS_BIT_KHR`, or `VK_SHADER_STAGE_CALLABLE_BIT_KHR` shader in it.

- **VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR** specifies a shader group that only hits triangles and must not contain an intersection shader, only closest hit and any-hit shaders.

- **VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR** specifies a shader group that only intersects with custom geometry and must contain an intersection shader and may contain closest hit and any-hit shaders.
For current group types, the hit group type could be inferred from the presence or absence of the intersection shader, but we provide the type explicitly for future hit groups that do not have that property.

The `VkRayTracingPipelineInterfaceCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_ray_tracing_pipeline
typedef struct VkRayTracingPipelineInterfaceCreateInfoKHR {
    VkStructureType  sType;
    const void*      pNext;
    uint32_t          maxPipelineRayPayloadSize;
    uint32_t          maxPipelineRayHitAttributeSize;
} VkRayTracingPipelineInterfaceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxPipelineRayPayloadSize` is the maximum payload size in bytes used by any shader in the pipeline.
- `maxPipelineRayHitAttributeSize` is the maximum attribute structure size in bytes used by any shader in the pipeline.

`maxPipelineRayPayloadSize` is calculated as the maximum number of bytes used by any block declared in the `RayPayloadKHR` or `IncomingRayPayloadKHR` storage classes. `maxPipelineRayHitAttributeSize` is calculated as the maximum number of bytes used by any block declared in the `HitAttributeKHR` storage class. As variables in these storage classes do not have explicit offsets, the size should be calculated as if each variable has a scalar alignment equal to the largest scalar alignment of any of the block’s members.

There is no explicit upper limit for `maxPipelineRayPayloadSize`, but in practice it should be kept as small as possible. Similar to invocation local memory, it must be allocated for each shader invocation and for devices which support many simultaneous invocations, this storage can rapidly be exhausted, resulting in failure.

Valid Usage

- `VUID-VkRayTracingPipelineInterfaceCreateInfoKHR-maxPipelineRayHitAttributeSize-03605` `maxPipelineRayHitAttributeSize` must be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxRayHitAttributeSize`
Valid Usage (Implicit)

- **VUID-VkRayTracingPipelineInterfaceCreateInfoKHR-sType-sType**
  
  *sType* must be **VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_INTERFACE_CREATE_INFO_KHR**

- **VUID-VkRayTracingPipelineInterfaceCreateInfoKHR-pNext-pNext**
  
  *pNext* must be **NULL**

To query the opaque handles of shaders in the ray tracing pipeline, call:

```c
// Provided by VK_KHR_ray_tracing_pipeline
VkResult vkGetRayTracingShaderGroupHandlesKHR(
    VkDevice                                    device,
    VkPipeline                                  pipeline,
    uint32_t                                    firstGroup,
    uint32_t                                    groupCount,
    size_t                                      dataSize,
    void*                                       pData);
```

or the equivalent command

```c
// Provided by VK_NV_ray_tracing
VkResult vkGetRayTracingShaderGroupHandlesNV(
    VkDevice                                    device,
    VkPipeline                                  pipeline,
    uint32_t                                    firstGroup,
    uint32_t                                    groupCount,
    size_t                                      dataSize,
    void*                                       pData);
```

- **device** is the logical device containing the ray tracing pipeline.
- **pipeline** is the ray tracing pipeline object containing the shaders.
- **firstGroup** is the index of the first group to retrieve a handle for from the **VkRayTracingPipelineCreateInfoKHR::pGroups** or **VkRayTracingPipelineCreateInfoNV::pGroups** array.
- **groupCount** is the number of shader handles to retrieve.
- **dataSize** is the size in bytes of the buffer pointed to by **pData**.
- **pData** is a pointer to a user-allocated buffer where the results will be written.
Valid Usage

- VUID-vkGetRayTracingShaderGroupHandlesKHR-pipeline-04619
  pipeline must be a ray tracing pipeline

- VUID-vkGetRayTracingShaderGroupHandlesKHR-firstGroup-04050
  firstGroup must be less than the number of shader groups in pipeline

- VUID-vkGetRayTracingShaderGroupHandlesKHR-firstGroup-02419
  The sum of firstGroup and groupCount must be less than or equal to the number of shader groups in pipeline

- VUID-vkGetRayTracingShaderGroupHandlesKHR-dataSize-02420
  dataSize must be at least VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleSize × groupCount

- VUID-vkGetRayTracingShaderGroupHandlesKHR-pipeline-03482
  pipeline must have not been created with VK_PIPELINE_CREATE_LIBRARY_BIT_KHR

Valid Usage (Implicit)

- VUID-vkGetRayTracingShaderGroupHandlesKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetRayTracingShaderGroupHandlesKHR-pipeline-parameter
  pipeline must be a valid VkPipeline handle

- VUID-vkGetRayTracingShaderGroupHandlesKHR-pData-parameter
  pData must be a valid pointer to an array of dataSize bytes

- VUID-vkGetRayTracingShaderGroupHandlesKHR-dataSize-arraylength
  dataSize must be greater than 0

- VUID-vkGetRayTracingShaderGroupHandlesKHR-pipeline-parent
  pipeline must have been created, allocated, or retrieved from device

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To query the optional capture handle information of shaders in the ray tracing pipeline, call:
Provided by VK_KHR_ray_tracing_pipeline

```c
VkResult vkGetRayTracingCaptureReplayShaderGroupHandlesKHR(
    VkDevice                                    device,
    VkPipeline                                  pipeline,
    uint32_t                                    firstGroup,
    uint32_t                                    groupCount,
    size_t                                      dataSize,
    void*                                       pData);
```

- **device** is the logical device containing the ray tracing pipeline.
- **pipeline** is the ray tracing pipeline object containing the shaders.
- **firstGroup** is the index of the first group to retrieve a handle for from the `VkRayTracingPipelineCreateInfoKHR::pGroups` array.
- **groupCount** is the number of shader handles to retrieve.
- **dataSize** is the size in bytes of the buffer pointed to by `pData`.
- **pData** is a pointer to a user-allocated buffer where the results will be written.

### Valid Usage

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-pipeline-04620
  - pipeline must be a ray tracing pipeline

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-firstGroup-04051
  - firstGroup must be less than the number of shader groups in pipeline

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-firstGroup-03483
  - The sum of `firstGroup` and `groupCount` must be less than or equal to the number of shader groups in pipeline

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-dataSize-03484
  - dataSize must be at least `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleCaptureReplaySize` × `groupCount`

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-rayTracingPipelineShaderGroupHandleCaptureReplay-03606
  - `VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipelineShaderGroupHandleCaptureReplay` must be enabled to call this function

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-pipeline-03607
  -pipeline must have been created with a flags that included `VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR`
Valid Usage (Implicit)

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-device-parameter
  
  *device* must be a valid *VkDevice* handle

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-pipeline-parameter
  
  *pipeline* must be a valid *VkPipeline* handle

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-pData-parameter
  
  *pData* must be a valid pointer to an array of *dataSize* bytes

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-dataSize-arraylength
  
  *dataSize* must be greater than 0

- VUID-vkGetRayTracingCaptureReplayShaderGroupHandlesKHR-pipeline-parent
  
  *pipeline* must have been created, allocated, or retrieved from *device*

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Ray tracing pipelines can contain more shaders than a graphics or compute pipeline, so to allow parallel compilation of shaders within a pipeline, an application can choose to defer compilation until a later point in time.

To compile a deferred shader in a pipeline call:

```c
// Provided by VK_NV_ray_tracing
VkResult vkCompileDeferredNV(
    VkDevice device,
    VkPipeline pipeline,
    uint32_t shader);
```

- *device* is the logical device containing the ray tracing pipeline.
- *pipeline* is the ray tracing pipeline object containing the shaders.
- *shader* is the index of the shader to compile.
Valid Usage

- VUID-vkCompileDeferredNV-pipeline-04621
  pipeline must be a ray tracing pipeline

- VUID-vkCompileDeferredNV-pipeline-02237
  pipeline must have been created with VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV

- VUID-vkCompileDeferredNV-shader-02238
  shader must not have been called as a deferred compile before

Valid Usage (Implicit)

- VUID-vkCompileDeferredNV-device-parameter
  device must be a valid VkDevice handle

- VUID-vkCompileDeferredNV-pipeline-parameter
  pipeline must be a valid VkPipeline handle

- VUID-vkCompileDeferredNV-pipeline-parent
  pipeline must have been created, allocated, or retrieved from device

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To query the pipeline stack size of shaders in a shader group in the ray tracing pipeline, call:

```c
// Provided by VK_KHR_ray_tracing_pipeline
VkDeviceSize vkGetRayTracingShaderGroupStackSizeKHR(
    VkDevice                                    device,
    VkPipeline                                  pipeline,
    uint32_t                                    group,
    VkShaderGroupShaderKHR                      groupShader);
```

- **device** is the logical device containing the ray tracing pipeline.
- **pipeline** is the ray tracing pipeline object containing the shaders groups.
- **group** is the index of the shader group to query.
- **groupShader** is the type of shader from the group to query.
The return value is the ray tracing pipeline stack size in bytes for the specified shader as called from the specified shader group.

### Valid Usage

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-pipeline-04622**
  
  *pipeline must be a ray tracing pipeline*

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-group-03608**
  
  *The value of group must be less than the number of shader groups in pipeline*

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-groupShader-03609**
  
  *The shader identified by groupShader in group must not be VK_SHADER_UNUSED_KHR*

### Valid Usage (Implicit)

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-device-parameter**
  
  *device must be a valid VkDevice handle*

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-pipeline-parameter**
  
  *pipeline must be a valid VkPipeline handle*

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-groupShader-parameter**
  
  *groupShader must be a valid VkShaderGroupShaderKHR value*

- **VUID-vkGetRayTracingShaderGroupStackSizeKHR-pipeline-parent**
  
  *pipeline must have been created, allocated, or retrieved from device*

Possible values of groupShader in vkGetRayTracingShaderGroupStackSizeKHR are:

```c
// Provided by VK_KHR_ray_tracing_pipeline
typedef enum VkShaderGroupShaderKHR {
    VK_SHADER_GROUP_SHADER_GENERAL_KHR = 0,
    VK_SHADER_GROUP_SHADER_CLOSEST_HIT_KHR = 1,
    VK_SHADER_GROUP_SHADER_ANY_HIT_KHR = 2,
    VK_SHADER_GROUP_SHADER_INTERSECTION_KHR = 3,
} VkShaderGroupShaderKHR;
```

- **VK_SHADER_GROUP_SHADER_GENERAL_KHR** uses the shader specified in the group with VkRayTracingShaderGroupCreateInfoKHR::generalShader

- **VK_SHADER_GROUP_SHADER_CLOSEST_HIT_KHR** uses the shader specified in the group with VkRayTracingShaderGroupCreateInfoKHR::closestHitShader

- **VK_SHADER_GROUP_SHADER_ANY_HIT_KHR** uses the shader specified in the group with VkRayTracingShaderGroupCreateInfoKHR::anyHitShader

- **VK_SHADER_GROUP_SHADER_INTERSECTION_KHR** uses the shader specified in the group with VkRayTracingShaderGroupCreateInfoKHR::intersectionShader
To set the dynamic stack size for a ray tracing pipeline call:

```c
// Provided by VK_KHR_ray_tracing_pipeline
void vkCmdSetRayTracingPipelineStackSizeKHR(
    VkCommandBuffer commandBuffer,
    uint32_t pipelineStackSize);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pipelineStackSize` is the stack size to use for subsequent ray tracing trace commands.

See [Ray Tracing Pipeline Stack](#) for more on computing `pipelineStackSize`.

### Valid Usage

- VUID-vkCmdSetRayTracingPipelineStackSizeKHR-pipelineStackSize-03610
  `pipelineStackSize` **must** be large enough for any dynamic execution through the shaders in the ray tracing pipeline used by a subsequent trace call

### Valid Usage (Implicit)

- VUID-vkCmdSetRayTracingPipelineStackSizeKHR-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetRayTracingPipelineStackSizeKHR-commandBuffer-recording
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdSetRayTracingPipelineStackSizeKHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support compute operations

- VUID-vkCmdSetRayTracingPipelineStackSizeKHR-renderpass
  This command **must** only be called outside of a render pass instance

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized
### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 10.4. Pipeline Destruction

To destroy a graphics or compute pipeline, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipeline(
    VkDevice device,
    VkPipeline pipeline,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the pipeline.
- **pipeline** is the handle of the pipeline to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyPipeline-pipeline-00765**
  
  All submitted commands that refer to pipeline must have completed execution

- **VUID-vkDestroyPipeline-pipeline-00766**
  
  If VkAllocationCallbacks were provided when pipeline was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyPipeline-pipeline-00767**
  
  If no VkAllocationCallbacks were provided when pipeline was created, pAllocator must be NULL
Valid Usage (Implicit)

- VUID-vkDestroyPipeline-device-parameter
device must be a valid VkDevice handle

- VUID-vkDestroyPipeline-pipeline-parameter
If pipeline is not VK_NULL_HANDLE, pipeline must be a valid VkPipeline handle

- VUID-vkDestroyPipeline-pAllocator-parameter
If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyPipeline-pipeline-parent
If pipeline is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to pipeline must be externally synchronized

10.5. Multiple Pipeline Creation

Multiple pipelines can be created simultaneously by passing an array of VkGraphicsPipelineCreateInfo, VkRayTracingPipelineCreateInfoKHR, VkRayTracingPipelineCreateInfoNV, or VkComputePipelineCreateInfo structures into the vkCreateGraphicsPipelines, vkCreateRayTracingPipelinesKHR, vkCreateRayTracingPipelinesNV, and vkCreateComputePipelines commands, respectively. Applications can group together similar pipelines to be created in a single call, and implementations are encouraged to look for reuse opportunities within a group-create.

When an application attempts to create many pipelines in a single command, it is possible that some subset may fail creation. In that case, the corresponding entries in the pPipelines output array will be filled with VK_NULL_HANDLE values. If any pipeline fails creation despite valid arguments (for example, due to out of memory errors), the VkResult code returned by vkCreate*Pipelines will indicate why. The implementation will attempt to create all pipelines, and only return VK_NULL_HANDLE values for those that actually failed.

If creation fails for a pipeline that had VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT set, pipelines at an index in the pPipelines array greater than or equal to that of the failing pipeline must be set to VK_NULL_HANDLE.

10.6. Pipeline Derivatives

A pipeline derivative is a child pipeline created from a parent pipeline, where the child and parent are expected to have much commonality. The goal of derivative pipelines is that they be cheaper to create using the parent as a starting point, and that it be more efficient (on either host or device) to switch/bind between children of the same parent.
A derivative pipeline is created by setting the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag in the VkPipelineCreateInfo structure. If this is set, then exactly one of basePipelineHandle or basePipelineIndex members of the structure must have a valid handle/index, and specifies the parent pipeline. If basePipelineHandle is used, the parent pipeline must have already been created. If basePipelineIndex is used, then the parent is being created in the same command. VK_NULL_HANDLE acts as the invalid handle for basePipelineHandle, and -1 is the invalid index for basePipelineIndex. If basePipelineIndex is used, the base pipeline must appear earlier in the array. The base pipeline must have been created with the VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT flag set.

## 10.7. Pipeline Cache

Pipeline cache objects allow the result of pipeline construction to be reused between pipelines and between runs of an application. Reuse between pipelines is achieved by passing the same pipeline cache object when creating multiple related pipelines. Reuse across runs of an application is achieved by retrieving pipeline cache contents in one run of an application, saving the contents, and using them to preinitialize a pipeline cache on a subsequent run. The contents of the pipeline cache objects are managed by the implementation. Applications can manage the host memory consumed by a pipeline cache object and control the amount of data retrieved from a pipeline cache object.

Pipeline cache objects are represented by VkPipelineCache handles:

```
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineCache)
```

To create pipeline cache objects, call:

```
// Provided by VK_VERSION_1_0
VkResult vkCreatePipelineCache(
  VkDevice device,
  const VkPipelineCacheCreateInfo* pCreateInfo,
  const VkAllocationCallbacks* pAllocator,
  VkPipelineCache* pPipelineCache);
```

- **device** is the logical device that creates the pipeline cache object.
- **pCreateInfo** is a pointer to a VkPipelineCacheCreateInfo structure containing initial parameters for the pipeline cache object.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pPipelineCache** is a pointer to a VkPipelineCache handle in which the resulting pipeline cache object is returned.
Note
Applications **can** track and manage the total host memory size of a pipeline cache object using the `pAllocator`. Applications **can** limit the amount of data retrieved from a pipeline cache object in `vkGetPipelineCacheData`. Implementations **should** not internally limit the total number of entries added to a pipeline cache object or the total host memory consumed.

Once created, a pipeline cache **can** be passed to the `vkCreateGraphicsPipelines`, `vkCreateRayTracingPipelinesKHR`, `vkCreateRayTracingPipelinesNV`, and `vkCreateComputePipelines` commands. If the pipeline cache passed into these commands is not `VK_NULL_HANDLE`, the implementation will query it for possible reuse opportunities and update it with new content. The use of the pipeline cache object in these commands is internally synchronized, and the same pipeline cache object **can** be used in multiple threads simultaneously.

If **flags** of `pCreateInfo` includes `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT`, all commands that modify the returned pipeline cache object **must** be externally synchronized.

Note
Implementations **should** make every effort to limit any critical sections to the actual accesses to the cache, which is expected to be significantly shorter than the duration of the `vkCreate*Pipelines` commands.

### Valid Usage (Implicit)

- **VUID-vkCreatePipelineCache-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkCreatePipelineCache-pCreateInfo-parameter**
  
  `pCreateInfo` must be a valid pointer to a valid `VkPipelineCacheCreateInfo` structure

- **VUID-vkCreatePipelineCache-pAllocator-parameter**
  
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreatePipelineCache-pPipelineCache-parameter**
  
  `pPipelineCache` must be a valid pointer to a `VkPipelineCache` handle

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
The `VkPipelineCacheCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineCacheCreateInfo {
    VkStructureType               sType;
    const void*                   pNext;
    VkPipelineCacheCreateFlags    flags;
    size_t                        initialDataSize;
    const void*                   pInitialData;
} VkPipelineCacheCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkPipelineCacheCreateFlagBits` specifying the behavior of the pipeline cache.
- **initialDataSize** is the number of bytes in `pInitialData`. If `initialDataSize` is zero, the pipeline cache will initially be empty.
- **pInitialData** is a pointer to previously retrieved pipeline cache data. If the pipeline cache data is incompatible (as defined below) with the device, the pipeline cache will be initially empty. If `initialDataSize` is zero, `pInitialData` is ignored.

### Valid Usage

- **VUID-VkPipelineCacheCreateInfo-initialDataSize-00768**
  If `initialDataSize` is not 0, it **must** be equal to the size of `pInitialData`, as returned by `vkGetPipelineCacheData` when `pInitialData` was originally retrieved.

- **VUID-VkPipelineCacheCreateInfo-initialDataSize-00769**
  If `initialDataSize` is not 0, `pInitialData` **must** have been retrieved from a previous call to `vkGetPipelineCacheData`.

- **VUID-VkPipelineCacheCreateInfo-pipelineCreationCacheControl-02892**
  If the `pipelineCreationCacheControl` feature is not enabled, **flags** **must** not include `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT`
Valid Usage (Implicit)

- **VUID-VkPipelineCacheCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO`

- **VUID-VkPipelineCacheCreateInfo-pNext-pNext**
  
  `pNext` must be `NULL`

- **VUID-VkPipelineCacheCreateInfo-flags-parameter**
  
  `flags` must be a valid combination of `VkPipelineCacheCreateFlagBits` values

- **VUID-VkPipelineCacheCreateInfo-initialData-parameter**
  
  If `initialDataSize` is not 0, `pInitialData` must be a valid pointer to an array of `initialDataSize` bytes

```
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineCacheCreateFlags;
```

`VkPipelineCacheCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineCacheCreateFlagBits` values.

Possible values of the `flags` member of `VkPipelineCacheCreateInfo`, specifying the behavior of the pipeline cache, are:

```
// Provided by VK_VERSION_1_0
typedef enum VkPipelineCacheCreateFlagBits {
    // Provided by VK_EXT_pipeline_creation_cache_control
    VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT = 0x00000001,
} VkPipelineCacheCreateFlagBits;
```

- `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT` specifies that all commands that modify the created `VkPipelineCache` will be externally synchronized. When set, the implementation may skip any unnecessary processing needed to support simultaneous modification from multiple threads where allowed.

Pipeline cache objects can be merged using the command:

```
// Provided by VK_VERSION_1_0
VkResult vkMergePipelineCaches(
    VkDevice                                    device,
    VkPipelineCache                             dstCache,
    uint32_t                                    srcCacheCount,
    const VkPipelineCache*                      pSrcCaches);
```

- `device` is the logical device that owns the pipeline cache objects.
- `dstCache` is the handle of the pipeline cache to merge results into.
• `srcCacheCount` is the length of the `pSrcCaches` array.
• `pSrcCaches` is a pointer to an array of pipeline cache handles, which will be merged into `dstCache`. The previous contents of `dstCache` are included after the merge.

**Note**
The details of the merge operation are implementation dependent, but implementations **should** merge the contents of the specified pipelines and prune duplicate entries.

---

**Valid Usage**

- VUID-vkMergePipelineCaches-dstCache-00770
  - `dstCache` must not appear in the list of source caches

---

**Valid Usage (Implicit)**

- VUID-vkMergePipelineCaches-device-parameter
  - `device` must be a valid `VkDevice` handle

- VUID-vkMergePipelineCaches-dstCache-parameter
  - `dstCache` must be a valid `VkPipelineCache` handle

- VUID-vkMergePipelineCaches-pSrcCaches-parameter
  - `pSrcCaches` must be a valid pointer to an array of `srcCacheCount` valid `VkPipelineCache` handles

- VUID-vkMergePipelineCaches-srcCacheCount-arraylength
  - `srcCacheCount` must be greater than 0

- VUID-vkMergePipelineCaches-dstCache-parent
  - `dstCache` must have been created, allocated, or retrieved from `device`

- VUID-vkMergePipelineCaches-pSrcCaches-parent
  - Each element of `pSrcCaches` must have been created, allocated, or retrieved from `device`

---

**Host Synchronization**

- Host access to `dstCache` must be externally synchronized
Data can be retrieved from a pipeline cache object using the command:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetPipelineCacheData(
    VkDevice device,            // Logical device
    VkPipelineCache pipelineCache, // Pipeline cache
    size_t* pDataSize,            // Pointer to value related to amount of data
    void* pData);                 // Buffer or NULL
```

- `device` is the logical device that owns the pipeline cache.
- `pipelineCache` is the pipeline cache to retrieve data from.
- `pDataSize` is a pointer to a `size_t` value related to the amount of data in the pipeline cache, as described below.
- `pData` is either `NULL` or a pointer to a buffer.

If `pData` is `NULL`, then the maximum size of the data that can be retrieved from the pipeline cache, in bytes, is returned in `pDataSize`. Otherwise, `pDataSize` must point to a variable set by the user to the size of the buffer, in bytes, pointed to by `pData`, and on return the variable is overwritten with the amount of data actually written to `pData`. If `pDataSize` is less than the maximum size that can be retrieved by the pipeline cache, at most `pDataSize` bytes will be written to `pData`, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all of the pipeline cache was returned.

Any data written to `pData` is valid and can be provided as the `pInitialData` member of the `VkPipelineCacheCreateInfo` structure passed to `vkCreatePipelineCache`.

Two calls to `vkGetPipelineCacheData` with the same parameters must retrieve the same data unless a command that modifies the contents of the cache is called between them.

Applications can store the data retrieved from the pipeline cache, and use these data, possibly in a future run of the application, to populate new pipeline cache objects. The results of pipeline compiles, however, may depend on the vendor ID, device ID, driver version, and other details of the device. To enable applications to detect when previously retrieved data is incompatible with the device, the initial bytes written to `pData` must be a header consisting of the following members:

Table 13. Layout for pipeline cache header version `VK_PIPELINE_CACHE_HEADER_VERSION_ONE`
<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>length in bytes of the entire pipeline cache header written as a stream of bytes, with the least significant byte first</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>a <code>VkPipelineCacheHeaderVersion</code> value written as a stream of bytes, with the least significant byte first</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>a vendor ID equal to <code>VkPhysicalDeviceProperties::vendorID</code> written as a stream of bytes, with the least significant byte first</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>a device ID equal to <code>VkPhysicalDeviceProperties::deviceID</code> written as a stream of bytes, with the least significant byte first</td>
</tr>
<tr>
<td>16</td>
<td><code>VK_UUID_SIZE</code></td>
<td>a pipeline cache ID equal to <code>VkPhysicalDeviceProperties::pipelineCacheUUID</code></td>
</tr>
</tbody>
</table>

The first four bytes encode the length of the entire pipeline cache header, in bytes. This value includes all fields in the header including the pipeline cache version field and the size of the length field.

The next four bytes encode the pipeline cache version, as described for `VkPipelineCacheHeaderVersion`. A consumer of the pipeline cache should use the cache version to interpret the remainder of the cache header.

If `pDataSize` is less than what is necessary to store this header, nothing will be written to `pData` and zero will be written to `pDataSize`.

### Valid Usage (Implicit)

- **VUID-vkGetPipelineCacheData-device-parameter**
  
  *device* must be a valid `VkDevice` handle

- **VUID-vkGetPipelineCacheData-pipelineCache-parameter**
  
  *pipelineCache* must be a valid `VkPipelineCache` handle

- **VUID-vkGetPipelineCacheData-pDataSize-parameter**
  
  *pDataSize* must be a valid pointer to a `size_t` value

- **VUID-vkGetPipelineCacheData-pData-parameter**
  
  If the value referenced by `pDataSize` is not 0, and `pData` is not NULL, `pData` must be a valid pointer to an array of `pDataSize` bytes

- **VUID-vkGetPipelineCacheData-pipelineCache-parent**
  
  *pipelineCache* must have been created, allocated, or retrieved from *device*
Return Codes

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Possible values of the second group of four bytes in the header returned by `vkGetPipelineCacheData`, encoding the pipeline cache version, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineCacheHeaderVersion {
    VK_PIPELINE_CACHE_HEADER_VERSION_ONE = 1,
} VkPipelineCacheHeaderVersion;
```

- **VK_PIPELINE_CACHE_HEADER_VERSION_ONE** specifies version one of the pipeline cache.

To destroy a pipeline cache, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipelineCache(
    VkDevice device,                  // Provided by VK_VERSION_1_0
    VkPipelineCache pipelineCache,   // Provided by VK_VERSION_1_0
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the pipeline cache object.
- **pipelineCache** is the handle of the pipeline cache to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- **VUID-vkDestroyPipelineCache-pipelineCache-00771**
  If `VkAllocationCallbacks` were provided when `pipelineCache` was created, a compatible set of callbacks **must** be provided here.
- **VUID-vkDestroyPipelineCache-pipelineCache-00772**
  If no `VkAllocationCallbacks` were provided when `pipelineCache` was created, `pAllocator` **must** be `NULL`. 
Valid Usage (Implicit)

- **VUID-vkDestroyPipelineCache-device-parameter**
  
  *device* must be a valid *VkDevice* handle

- **VUID-vkDestroyPipelineCache-pipelineCache-parameter**
  
  If *pipelineCache* is not *VK_NULL_HANDLE*, *pipelineCache* must be a valid *VkPipelineCache* handle

- **VUID-vkDestroyPipelineCache-pAllocator-parameter**
  
  If *pAllocator* is not *NULL*, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkDestroyPipelineCache-pipelineCache-parent**
  
  If *pipelineCache* is a valid handle, it must have been created, allocated, or retrieved from *device*

Host Synchronization

- Host access to *pipelineCache* must be externally synchronized

10.8. Specialization Constants

Specialization constants are a mechanism whereby constants in a SPIR-V module can have their constant value specified at the time the *VkPipeline* is created. This allows a SPIR-V module to have constants that can be modified while executing an application that uses the Vulkan API.

**Note**

Specialization constants are useful to allow a compute shader to have its local workgroup size changed at runtime by the user, for example.

Each *VkPipelineShaderStageCreateInfo* structure contains a *pSpecializationInfo* member, which can be *NULL* to indicate no specialization constants, or point to a *VkSpecializationInfo* structure.

The *VkSpecializationInfo* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSpecializationInfo {
    uint32_t           mapEntryCount;
    const VkSpecializationMapEntry* pMapEntries;
    size_t             dataSize;
    const void*        pData;
} VkSpecializationInfo;
```

- **mapEntryCount** is the number of entries in the *pMapEntries* array.
- **pMapEntries** is a pointer to an array of *VkSpecializationMapEntry* structures which map constant
IDs to offsets in `pData`.

- `dataSize` is the byte size of the `pData` buffer.
- `pData` contains the actual constant values to specialize with.

`pMapEntries` is a pointer to a `VkSpecializationMapEntry` structure.

### Valid Usage

- **VUID-VkSpecializationInfo-offset-00773**
  The offset member of each element of `pMapEntries` **must** be less than `dataSize`

- **VUID-VkSpecializationInfo-pMapEntries-00774**
  The size member of each element of `pMapEntries` **must** be less than or equal to `dataSize` minus offset

### Valid Usage (Implicit)

- **VUID-VkSpecializationInfo-pMapEntries-parameter**
  If `mapEntryCount` is not 0, `pMapEntries` **must** be a valid pointer to an array of `mapEntryCount` valid `VkSpecializationMapEntry` structures

- **VUID-VkSpecializationInfo-pData-parameter**
  If `dataSize` is not 0, `pData` **must** be a valid pointer to an array of `dataSize` bytes

The `VkSpecializationMapEntry` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSpecializationMapEntry {
    uint32_t constantID;
    uint32_t offset;
    size_t size;
} VkSpecializationMapEntry;
```

- `constantID` is the ID of the specialization constant in SPIR-V.
- `offset` is the byte offset of the specialization constant value within the supplied data buffer.
- `size` is the byte size of the specialization constant value within the supplied data buffer.

If a `constantID` value is not a specialization constant ID used in the shader, that map entry does not affect the behavior of the pipeline.
Valid Usage

- VUID-VkSpecializationMapEntry-constantID-00776

For a constantID specialization constant declared in a shader, size must match the byte size of the constantID. If the specialization constant is of type boolean, size must be the byte size of VkBool32

In human readable SPIR-V:

```
OpDecorate %x SpecId 13 ; decorate .x component of WorkgroupSize with ID 13
OpDecorate %y SpecId 42 ; decorate .y component of WorkgroupSize with ID 42
OpDecorate %z SpecId 3 ; decorate .z component of WorkgroupSize with ID 3
OpDecorate %wgsize BuiltIn WorkgroupSize ; decorate WorkgroupSize onto constant
%32 = OpTypeInt 32 0 ; declare an unsigned 32-bit type
%uvec3 = OpTypeVector %32 3 ; declare a 3 element vector type of unsigned 32-bit
%x = OpSpecConstant %32 1 ; declare the .x component of WorkgroupSize
%y = OpSpecConstant %32 1 ; declare the .y component of WorkgroupSize
%z = OpSpecConstant %32 1 ; declare the .z component of WorkgroupSize
%wgsize = OpSpecConstantComposite %uvec3 %x %y %z ; declare WorkgroupSize
```

From the above we have three specialization constants, one for each of the x, y & z elements of the WorkgroupSize vector.

Now to specialize the above via the specialization constants mechanism:
```c
const VkSpecializationMapEntry entries[] = {
    {
        13, // constantID
        0 * sizeof(uint32_t), // offset
        sizeof(uint32_t) // size
    },
    {
        42, // constantID
        1 * sizeof(uint32_t), // offset
        sizeof(uint32_t) // size
    },
    {
        3, // constantID
        2 * sizeof(uint32_t), // offset
        sizeof(uint32_t) // size
    }
};

const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info = {
    3, // mapEntryCount
    entries, // pMapEntries
    3 * sizeof(uint32_t), // dataSize
    data, // pData
};
```

Then when calling `vkCreateComputePipelines`, and passing the `VkSpecializationInfo` we defined as the `pSpecializationInfo` parameter of `VkPipelineShaderStageCreateInfo`, we will create a compute pipeline with the runtime specified local workgroup size.

Another example would be that an application has a SPIR-V module that has some platform-dependent constants they wish to use.

In human readable SPIR-V:

```
OpDecorate %1 SpecId 0 ; decorate our signed 32-bit integer constant
OpDecorate %2 SpecId 12 ; decorate our 32-bit floating-point constant
%32 = OpTypeInt 32 1 ; declare a signed 32-bit type
%float = OpTypeFloat 32 ; declare a 32-bit floating-point type
%1 = OpSpecConstant %i32 -1 ; some signed 32-bit integer constant
%2 = OpSpecConstant %float 0.5 ; some 32-bit floating-point constant
```

From the above we have two specialization constants, one is a signed 32-bit integer and the second is a 32-bit floating-point value.
Now to specialize the above via the specialization constants mechanism:

```c
struct SpecializationData {
    int32_t data0;
    float data1;
};

const VkSpecializationMapEntry entries[] = {
    { 0, offsetof(SpecializationData, data0), sizeof(SpecializationData::data0) },
    { 12, offsetof(SpecializationData, data1), sizeof(SpecializationData::data1) }
};

SpecializationData data;
data.data0 = -42; // set the data for the 32-bit integer
data.data1 = 42.0f; // set the data for the 32-bit floating-point

const VkSpecializationInfo info = {
    2, entries, sizeof(data), &data,
};
```

It is legal for a SPIR-V module with specializations to be compiled into a pipeline where no specialization info was provided. SPIR-V specialization constants contain default values such that if a specialization is not provided, the default value will be used. In the examples above, it would be valid for an application to only specialize some of the specialization constants within the SPIR-V module, and let the other constants use their default values encoded within the OpSpecConstant declarations.

### 10.9. Pipeline Libraries

A pipeline library is a special pipeline that was created using the `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR` and cannot be bound, instead it defines a set of pipeline state which can be linked into other pipelines. For ray tracing pipelines this includes shaders and shader groups. The application must maintain the lifetime of pipeline libraries based on the pipelines that link with it. A pipeline library is considered in-use, as long as one of the linking pipelines is in-use.

This linkage is achieved by using the following structure within the appropriate creation
mechanisms:

The `VkPipelineLibraryCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_pipeline_library
typedef struct VkPipelineLibraryCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t libraryCount;
    const VkPipeline* pLibraries;
} VkPipelineLibraryCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `libraryCount` is the number of pipeline libraries in `pLibraries`.
- `pLibraries` is an array of pipeline libraries to use when creating a pipeline.

**Valid Usage**

- VUID-VkPipelineLibraryCreateInfoKHR-pLibraries-03381
  
  Each element of `pLibraries` must have been created with `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR`.

**Valid Usage (Implicit)**

- VUID-VkPipelineLibraryCreateInfoKHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR`.

- VUID-VkPipelineLibraryCreateInfoKHR-pNext-pNext
  
  `pNext` must be `NULL`.

- VUID-VkPipelineLibraryCreateInfoKHR-pLibraries-parameter
  
  If `libraryCount` is not 0, `pLibraries` must be a valid pointer to an array of `libraryCount` valid `VkPipeline` handles.

### 10.10. Pipeline Binding

Once a pipeline has been created, it can be bound to the command buffer using the command:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindPipeline(
    VkCommandBuffer commandBuffer,
    VkPipelineBindPoint pipelineBindPoint,
    VkPipeline pipeline);
```
• **commandBuffer** is the command buffer that the pipeline will be bound to.

• **pipelineBindPoint** is a **VkPipelineBindPoint** value specifying to which bind point the pipeline is bound. Binding one does not disturb the others.

• **pipeline** is the pipeline to be bound.

Once bound, a pipeline binding affects subsequent commands that interact with the given pipeline type in the command buffer until a different pipeline of the same type is bound to the bind point. Commands that do not interact with the given pipeline type **must** not be affected by the pipeline state.

• The pipeline bound to **VK_PIPELINE_BIND_POINT_COMPUTE** controls the behavior of all dispatching commands.

• The pipeline bound to **VK_PIPELINE_BIND_POINT_GRAPHICS** controls the behavior of all drawing commands.

• The pipeline bound to **VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR** controls the behavior of **vkCmdTraceRaysKHR** and **vkCmdTraceRaysIndirectKHR**.
Valid Usage

- VUID-vkCmdBindPipeline-pipelineBindPoint-00777
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, the `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.

- VUID-vkCmdBindPipeline-pipelineBindPoint-00778
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

- VUID-vkCmdBindPipeline-pipelineBindPoint-00779
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, pipeline must be a compute pipeline.

- VUID-vkCmdBindPipeline-pipelineBindPoint-00780
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, pipeline must be a graphics pipeline.

- VUID-vkCmdBindPipeline-pipelineBindPoint-00781
  If the variable multisample rate feature is not supported, pipeline is a graphics pipeline, the current subpass uses no attachments, and this is not the first call to this function with a graphics pipeline after transitioning to the current subpass, then the sample count specified by this pipeline must match that set in the previous pipeline.

- VUID-vkCmdBindPipeline-variableSampleLocations-01525
  If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE`, and pipeline is a graphics pipeline created with a `VkPipelineSampleLocationsStateCreateInfoEXT` structure having its `sampleLocationsEnable` member set to `VK_TRUE` but without `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` enabled then the current render pass instance must have been begun by specifying a `VkRenderPassSampleLocationsBeginInfoEXT` structure whose `pPostSubpassSampleLocations` member contains an element with a `subpassIndex` matching the current subpass index and the `sampleLocationsInfo` member of that element must match the `sampleLocationsInfo` specified in `VkPipelineSampleLocationsStateCreateInfoEXT` when the pipeline was created.

- VUID-vkCmdBindPipeline-None-02323
  This command must not be recorded when transform feedback is active.

- VUID-vkCmdBindPipeline-pipelineBindPoint-02391
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR`, the `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.

- VUID-vkCmdBindPipeline-pipelineBindPoint-02392
  If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR`, the pipeline must be a ray tracing pipeline.

- VUID-vkCmdBindPipeline-pipeline-03382
  The pipeline must not have been created with `VK_PIPELINE_CREATE_LIBRARY_BIT_KHR` set.

- VUID-vkCmdBindPipeline-commandBuffer-04808
  If `commandBuffer` is a secondary command buffer with `VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D` enabled and...
pipelineBindPoint is VK_PIPELINE_BIND_POINT_GRAPHICS, then the pipeline must have been created with VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT or VK_DYNAMIC_STATE_VIEWPORT, and VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT or VK_DYNAMIC_STATE_SCISSOR enabled.

- VUID-vkCmdBindPipeline-commandBuffer-04809
  If commandBuffer is a secondary command buffer with VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D enabled and pipelineBindPoint is VK_PIPELINE_BIND_POINT_GRAPHICS and pipeline was created with VkPipelineDiscardRectangleStateCreateInfoEXT structure and its discardRectangleCount member is not 0, then the pipeline must have been created with VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT enabled.

- VUID-vkCmdBindPipeline-pipelineBindPoint-04881
  If pipelineBindPoint is VK_PIPELINE_BIND_POINT_GRAPHICS and the provokingVertexModePerPipeline limit is VK_FALSE, then pipeline's VkPipelineRasterizationProvokingVertexStateCreateInfoEXT::provokingVertexMode must be the same as that of any other pipelines previously bound to this bind point within the current renderpass instance, including any pipeline already bound when beginning the renderpass instance.

**Valid Usage (Implicit)**

- VUID-vkCmdBindPipeline-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBindPipeline-pipelineBindPoint-parameter
pipelineBindPoint must be a valid VkPipelineBindPoint value

- VUID-vkCmdBindPipeline-pipeline-parameter
pipeline must be a valid VkPipeline handle

- VUID-vkCmdBindPipeline-commandBuffer-recording
commandBuffer must be in the recording state

- VUID-vkCmdBindPipeline-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdBindPipeline-commonparent
Both of commandBuffer, and pipeline must have been created, allocated, or retrieved from the same VkDevice

**Host Synchronization**

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
Possible values of `vkCmdBindPipeline::pipelineBindPoint`, specifying the bind point of a pipeline object, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPipelineBindPoint {
    VK_PIPELINE_BIND_POINT_GRAPHICS = 0,
    VK_PIPELINE_BIND_POINT_COMPUTE = 1,
    // Provided by VK_KHR_ray_tracing_pipeline
    VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR = 1000165000,
    // Provided by VK_NV_ray_tracing
    VK_PIPELINE_BIND_POINT_RAY_TRACING_NV = VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR,
} VkPipelineBindPoint;
```

- `VK_PIPELINE_BIND_POINT_COMPUTE` specifies binding as a compute pipeline.
- `VK_PIPELINE_BIND_POINT_GRAPHICS` specifies binding as a graphics pipeline.
- `VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR` specifies binding as a ray tracing pipeline.

For pipelines that were created with the support of multiple shader groups (see Graphics Pipeline Shader Groups), the regular `vkCmdBindPipeline` command will bind Shader Group 0. To explicitly bind a shader group use:

```c
// Provided by VK_NV_device_generated_commands
void vkCmdBindPipelineShaderGroupNV(
    VkCommandBuffer commandBuffer,           
    VkPipelineBindPoint pipelineBindPoint,   
    VkPipeline pipeline,                     
    uint32_t groupIndex);
```

- `commandBuffer` is the command buffer that the pipeline will be bound to.
- `pipelineBindPoint` is a `VkPipelineBindPoint` value specifying the bind point to which the pipeline will be bound.
- `pipeline` is the pipeline to be bound.
- `groupIndex` is the shader group to be bound.
Valid Usage

- VUID-vkCmdBindPipelineShaderGroupNV-groupIndex-02893
  
  \texttt{groupIndex} must be \(0\) or less than the effective \texttt{VkGraphicsPipelineShaderGroupsCreateInfoNV::groupCount} including the referenced pipelines.

- VUID-vkCmdBindPipelineShaderGroupNV-pipelineBindPoint-02894
  
  The \texttt{pipelineBindPoint} must be \texttt{VK_PIPELINE_BIND_POINT_GRAPHICS}.

- VUID-vkCmdBindPipelineShaderGroupNV-groupIndex-02895
  
  The same restrictions as \texttt{vkCmdBindPipeline} apply as if the bound pipeline was created only with the Shader Group from the \texttt{groupIndex} information.

- VUID-vkCmdBindPipelineShaderGroupNV-deviceGeneratedCommands-02896
  
  The \texttt{VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands} feature must be enabled.

Valid Usage (Implicit)

- VUID-vkCmdBindPipelineShaderGroupNV-commandBuffer-parameter
  
  \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle.

- VUID-vkCmdBindPipelineShaderGroupNV-pipelineBindPoint-parameter
  
  \texttt{pipelineBindPoint} must be a valid \texttt{VkPipelineBindPoint} value.

- VUID-vkCmdBindPipelineShaderGroupNV-pipeline-parameter
  
  \texttt{pipeline} must be a valid \texttt{VkPipeline} handle.

- VUID-vkCmdBindPipelineShaderGroupNV-commandBuffer-recording
  
  \texttt{commandBuffer} must be in the \texttt{recording} state.

- VUID-vkCmdBindPipelineShaderGroupNV-commandBuffer-cmdpool
  
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics, or compute operations.

- VUID-vkCmdBindPipelineShaderGroupNV-commonparent
  
  Both of \texttt{commandBuffer}, and \texttt{pipeline} must have been created, allocated, or retrieved from the same \texttt{VkDevice}.

Host Synchronization

- Host access to \texttt{commandBuffer} must be externally synchronized.

- Host access to the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must be externally synchronized.
10.11. Dynamic State

When a pipeline object is bound, any pipeline object state that is not specified as dynamic is applied to the command buffer state. Pipeline object state that is specified as dynamic is not applied to the command buffer state at this time. Instead, dynamic state can be modified at any time and persists for the lifetime of the command buffer, or until modified by another dynamic state setting command or another pipeline bind.

When a pipeline object is bound, the following applies to each state parameter:

- If the state is not specified as dynamic in the new pipeline object, then that command buffer state is overwritten by the state in the new pipeline object. Before any draw or dispatch call with this pipeline there must not have been any call to any of the corresponding dynamic state setting commands after this pipeline was bound.

- If the state is specified as dynamic in the new pipeline object, then that command buffer state is not disturbed. Before any draw or dispatch call with this pipeline there must have been at least one call to each of the corresponding dynamic state setting commands since the command buffer recording was begun, or the last bound pipeline object with that state specified as static, whichever was the latter.

Dynamic state that does not affect the result of operations can be left undefined.

Note

For example, if blending is disabled by the pipeline object state then the dynamic color blend constants do not need to be specified in the command buffer, even if this state is specified as dynamic in the pipeline object.

10.12. Pipeline Shader Information

When a pipeline is created, its state and shaders are compiled into zero or more device-specific executables, which are used when executing commands against that pipeline. To query the properties of these executables, call:
// Provided by VK_KHR_pipeline_executable_properties
VkResult vkGetPipelineExecutablePropertiesKHR(
    VkDevice device,
    const VkPipelineInfoKHR* pPipelineInfo,
    uint32_t* pExecutableCount,
    VkPipelineExecutablePropertiesKHR* pProperties);

*device* is the device that created the pipeline.

*pPipelineInfo* describes the pipeline being queried.

*pExecutableCount* is a pointer to an integer related to the number of pipeline executables available or queried, as described below.

*pProperties* is either NULL or a pointer to an array of VkPipelineExecutablePropertiesKHR structures.

If *pProperties* is NULL, then the number of executables associated with the pipeline is returned in *pExecutableCount*. Otherwise, *pExecutableCount* must point to a variable set by the user to the number of elements in the *pProperties* array, and on return the variable is overwritten with the number of structures actually written to *pProperties*. If *pExecutableCount* is less than the number of executables associated with the pipeline, at most *pExecutableCount* structures will be written, and VK_INCOMPLETE will be returned instead of VK_SUCCESS, to indicate that not all the available properties were returned.

---

### Valid Usage

- **VUID-vkGetPipelineExecutablePropertiesKHR-pipelineExecutableInfo-03270**
  pipelineExecutableInfo must be enabled

- **VUID-vkGetPipelineExecutablePropertiesKHR-pipeline-03271**
  pipeline member of *pPipelineInfo* must have been created with *device*

### Valid Usage (Implicit)

- **VUID-vkGetPipelineExecutablePropertiesKHR-device-parameter**
  *device* must be a valid VkDevice handle

- **VUID-vkGetPipelineExecutablePropertiesKHR-pPipelineInfo-parameter**
  *pPipelineInfo* must be a valid pointer to a valid VkPipelineInfoKHR structure

- **VUID-vkGetPipelineExecutablePropertiesKHR-pExecutableCount-parameter**
  *pExecutableCount* must be a valid pointer to a uint32_t value

- **VUID-vkGetPipelineExecutablePropertiesKHR-pProperties-parameter**
  If the value referenced by *pExecutableCount* is not 0, and *pProperties* is not NULL, *pProperties* must be a valid pointer to an array of *pExecutableCount* VkPipelineExecutablePropertiesKHR structures
Return Codes

Success

• VK_SUCCESS
• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkPipelineInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_pipeline_executable_properties
typedef struct VkPipelineInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkPipeline         pipeline;
} VkPipelineInfoKHR;
```

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `pipeline` is a `VkPipeline` handle.

Valid Usage (Implicit)

• VUID-VkPipelineInfoKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_INFO_KHR`

• VUID-VkPipelineInfoKHR-pNext-pNext
  `pNext` must be `NULL`

• VUID-VkPipelineInfoKHR-pipeline-parameter
  `pipeline` must be a valid `VkPipeline` handle

The `VkPipelineExecutablePropertiesKHR` structure is defined as:
// Provided by VK_KHR_pipeline_executable_properties

typedef struct VkPipelineExecutablePropertiesKHR {
    VkStructureType       sType;
    void*                 pNext;
    VkShaderStageFlags    stages;
    char                   name[VK_MAX_DESCRIPTION_SIZE];
    char                   description[VK_MAX_DESCRIPTION_SIZE];
    uint32_t              subgroupSize;
} VkPipelineExecutablePropertiesKHR;

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **stages** is a bitmask of **VkShaderStageFlagBits** indicating which shader stages (if any) were principally used as inputs to compile this pipeline executable.

• **name** is an array of **VK_MAX_DESCRIPTION_SIZE** char containing a null-terminated UTF-8 string which is a short human readable name for this executable.

• **description** is an array of **VK_MAX_DESCRIPTION_SIZE** char containing a null-terminated UTF-8 string which is a human readable description for this executable.

• **subgroupSize** is the subgroup size with which this executable is dispatched.

The **stages** field **may** be zero or it **may** contain one or more bits describing the stages principally used to compile this pipeline. Not all implementations have a 1:1 mapping between shader stages and pipeline executables and some implementations **may** reduce a given shader stage to fixed function hardware programming such that no executable is available. No guarantees are provided about the mapping between shader stages and pipeline executables and **stages should** be considered a best effort hint. Because the application **cannot** rely on the **stages** field to provide an exact description, **name** and **description** provide a human readable name and description which more accurately describes the given pipeline executable.

**Valid Usage (Implicit)**

- VUID-VkPipelineExecutablePropertiesKHR-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_PROPERTIES_KHR**

- VUID-VkPipelineExecutablePropertiesKHR-pNext-pNext
  
  **pNext** must be **NULL**

Each pipeline executable **may** have a set of statistics associated with it that are generated by the pipeline compilation process. These statistics **may** include things such as instruction counts, amount of spilling (if any), maximum number of simultaneous threads, or anything else which **may** aid developers in evaluating the expected performance of a shader. To query the compile-time statistics associated with a pipeline executable, call:
// Provided by VK_KHR_pipeline_executable_properties

VkResult vkGetPipelineExecutableStatisticsKHR(
    VkDevice device, 
    const VkPipelineExecutableInfoKHR* pExecutableInfo, 
    uint32_t* pStatisticCount, 
    VkPipelineExecutableStatisticKHR* pStatistics);

- **device** is the device that created the pipeline.
- **pExecutableInfo** describes the pipeline executable being queried.
- **pStatisticCount** is a pointer to an integer related to the number of statistics available or queried, as described below.
- **pStatistics** is either NULL or a pointer to an array of VkPipelineExecutableStatisticKHR structures.

If **pStatistics** is NULL, then the number of statistics associated with the pipeline executable is returned in **pStatisticCount**. Otherwise, **pStatisticCount** must point to a variable set by the user to the number of elements in the **pStatistics** array, and on return the variable is overwritten with the number of structures actually written to **pStatistics**. If **pStatisticCount** is less than the number of statistics associated with the pipeline executable, at most **pStatisticCount** structures will be written, and **VK_INCOMPLETE** will be returned instead of **VK_SUCCESS**, to indicate that not all the available statistics were returned.

### Valid Usage

- **VUID-vkGetPipelineExecutableStatisticsKHR-pipelineExecutableInfo-03272**
  **pipelineExecutableInfo** must be enabled

- **VUID-vkGetPipelineExecutableStatisticsKHR-pipeline-03273**
  **pipeline** member of **pExecutableInfo** must have been created with **device**

- **VUID-vkGetPipelineExecutableStatisticsKHR-pipeline-03274**
  **pipeline** member of **pExecutableInfo** must have been created with **VK_PIPELINE_CREATE_CAPTURE_STATISTICS_BIT_KHR**
**Valid Usage (Implicit)**

- **VUID-vkGetPipelineExecutableStatisticsKHR-device-parameter**
  
  *device must be a valid VkDevice handle*

- **VUID-vkGetPipelineExecutableStatisticsKHR-pExecutableInfo-parameter**
  
  *pExecutableInfo must be a valid pointer to a valid VkPipelineExecutableInfoKHR structure*

- **VUID-vkGetPipelineExecutableStatisticsKHR-pStatisticCount-parameter**
  
  *pStatisticCount must be a valid pointer to a uint32_t value*

- **VUID-vkGetPipelineExecutableStatisticsKHR-pStatistics-parameter**
  
  If the value referenced by pStatisticCount is not 0, and pStatistics is not NULL, pStatistics must be a valid pointer to an array of pStatisticCount VkPipelineExecutableStatisticKHR structures

**Return Codes**

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkPipelineExecutableInfoKHR structure is defined as:

```c
// Provided by VK_KHR_pipeline_executable_properties
typedef struct VkPipelineExecutableInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkPipeline pipeline;
    uint32_t executableIndex;
} VkPipelineExecutableInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `pipeline` is the pipeline to query.
- `executableIndex` is the index of the executable to query in the array of executable properties returned by vkGetPipelineExecutablePropertiesKHR.
Valid Usage

- VUID-VkPipelineExecutableInfoKHR-executableIndex-03275
  executableIndex must be less than the number of executables associated with pipeline as returned in the pExecutableCount parameter of vkGetPipelineExecutablePropertiesKHR

Valid Usage (Implicit)

- VUID-VkPipelineExecutableInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INFO_KHR

- VUID-VkPipelineExecutableInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkPipelineExecutableInfoKHR-pipeline-parameter
  pipeline must be a valid VkPipeline handle

The VkPipelineExecutableStatisticKHR structure is defined as:

```c
// Provided by VK_KHR_pipeline_executable_properties
typedef struct VkPipelineExecutableStatisticKHR {
    VkStructureType sType;
    void* pNext;
    char name[VK_MAX_DESCRIPTION_SIZE];
    char description[VK_MAX_DESCRIPTION_SIZE];
    VkPipelineExecutableStatisticFormatKHR format;
    VkPipelineExecutableStatisticValueKHR value;
} VkPipelineExecutableStatisticKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **name** is an array of VK_MAX_DESCRIPTION_SIZE char containing a null-terminated UTF-8 string which is a short human readable name for this statistic.
- **description** is an array of VK_MAX_DESCRIPTION_SIZE char containing a null-terminated UTF-8 string which is a human readable description for this statistic.
- **format** is a VkPipelineExecutableStatisticFormatKHR value specifying the format of the data found in **value**.
- **value** is the value of this statistic.
Valid Usage (Implicit)

- **VUID-VkPipelineExecutableStatisticKHR-sType-sType**
  
  Must be `VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_STATISTIC_KHR`

- **VUID-VkPipelineExecutableStatisticKHR-pNext-pNext**
  
  Must be NULL

The `VkPipelineExecutableStatisticFormatKHR` enum is defined as:

```c
// Provided by VK_KHR_pipeline_executable_properties
typedef enum VkPipelineExecutableStatisticFormatKHR {
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_BOOL32_KHR = 0,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_INT64_KHR = 1,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_UINT64_KHR = 2,
    VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_FLOAT64_KHR = 3,
} VkPipelineExecutableStatisticFormatKHR;
```

- **VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_BOOL32_KHR** specifies that the statistic is returned as a 32-bit boolean value which **must** be either `VK_TRUE` or `VK_FALSE` and **should** be read from the `b32` field of `VkPipelineExecutableStatisticValueKHR`.

- **VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_INT64_KHR** specifies that the statistic is returned as a signed 64-bit integer and **should** be read from the `i64` field of `VkPipelineExecutableStatisticValueKHR`.

- **VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_UINT64_KHR** specifies that the statistic is returned as an unsigned 64-bit integer and **should** be read from the `u64` field of `VkPipelineExecutableStatisticValueKHR`.

- **VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_FLOAT64_KHR** specifies that the statistic is returned as a 64-bit floating-point value and **should** be read from the `f64` field of `VkPipelineExecutableStatisticValueKHR`.

The `VkPipelineExecutableStatisticValueKHR` union is defined as:

```c
// Provided by VK_KHR_pipeline_executable_properties
typedef union VkPipelineExecutableStatisticValueKHR {
    VkBool32    b32;
    int64_t     i64;
    uint64_t    u64;
    double      f64;
} VkPipelineExecutableStatisticValueKHR;
```

- **b32** is the 32-bit boolean value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_BOOL32_KHR`.

- **i64** is the signed 64-bit integer value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_INT64_KHR`.

Chapter 10. Pipelines | 623
• **u64** is the unsigned 64-bit integer value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_UINT64_KHR`.

• **f64** is the 64-bit floating-point value if the `VkPipelineExecutableStatisticFormatKHR` is `VK_PIPELINE_EXECUTABLE_STATISTIC_FORMAT_FLOAT64_KHR`.

Each pipeline executable may have one or more text or binary internal representations associated with it which are generated as part of the compile process. These may include the final shader assembly, a binary form of the compiled shader, or the shader compiler's internal representation at any number of intermediate compile steps. To query the internal representations associated with a pipeline executable, call:

```c
// Provided by VK_KHR_pipeline_executable_properties
VkResult vkGetPipelineExecutableInternalRepresentationsKHR(
    VkDevice                                    device,
    const VkPipelineExecutableInfoKHR*          pExecutableInfo,  
    uint32_t*                                   pInternalRepresentationCount,  
    VkPipelineExecutableInternalRepresentationKHR* pInternalRepresentations);
```

• **device** is the device that created the pipeline.

• **pExecutableInfo** describes the pipeline executable being queried.

• **pInternalRepresentationCount** is a pointer to an integer related to the number of internal representations available or queried, as described below.

• **pInternalRepresentations** is either **NULL** or a pointer to an array of `VkPipelineExecutableInternalRepresentationKHR` structures.

If `pInternalRepresentations` is **NULL**, then the number of internal representations associated with the pipeline executable is returned in `pInternalRepresentationCount`. Otherwise, `pInternalRepresentationCount` must point to a variable set by the user to the number of elements in the `pInternalRepresentations` array, and on return the variable is overwritten with the number of structures actually written to `pInternalRepresentations`. If `pInternalRepresentationCount` is less than the number of internal representations associated with the pipeline executable, at most `pInternalRepresentationCount` structures will be written, and **VK_INCOMPLETE** will be returned instead of **VK_SUCCESS**, to indicate that not all the available representations were returned.

While the details of the internal representations remain implementation dependent, the implementation should order the internal representations in the order in which they occur in the compile pipeline with the final shader assembly (if any) last.
Valid Usage

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-pipelineExecutableInfo-03276
  `pipelineExecutableInfo must be enabled`

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-pipeline-03277
  `pipeline member of pExecutableInfo must have been created with device`

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-pipeline-03278
  `pipeline member of pExecutableInfo must have been created with
  VK_PIPELINE_CREATE_CAPTURE_INTERNAL_REPRESENTATIONS_BIT_KHR`

Valid Usage (Implicit)

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-device-parameter
  `device must be a valid VkDevice handle`

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-pExecutableInfo-parameter
  `pExecutableInfo must be a valid pointer to a valid VkPipelineExecutableInfoKHR
  structure`

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-pInternalRepresentationCount-parameter
  `pInternalRepresentationCount must be a valid pointer to a uint32_t value`

- VUID-vkGetPipelineExecutableInternalRepresentationsKHR-pInternalRepresentations-parameter
  `If the value referenced by pInternalRepresentationCount is not 0, and
  pInternalRepresentations is not NULL, pInternalRepresentations must be a valid pointer to
  an array of pInternalRepresentationCount
  VkPipelineExecutableInternalRepresentationKHR structures`

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkPipelineExecutableInternalRepresentationKHR` structure is defined as:
// Provided by VK_KHR_pipeline_executable_properties

typedef struct VkPipelineExecutableInternalRepresentationKHR {
    VkStructureType    sType;
    void*              pNext;
    char               name[VK_MAX_DESCRIPTION_SIZE];
    char               description[VK_MAX_DESCRIPTION_SIZE];
    VkBool32           isText;
    size_t             dataSize;
    void*              pData;
} VkPipelineExecutableInternalRepresentationKHR;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- name is an array of VK_MAX_DESCRIPTION_SIZE char containing a null-terminated UTF-8 string which is a short human readable name for this internal representation.
- description is an array of VK_MAX_DESCRIPTION_SIZE char containing a null-terminated UTF-8 string which is a human readable description for this internal representation.
- isText specifies whether the returned data is text or opaque data. If isText is VK_TRUE then the data returned in pData is text and is guaranteed to be a null-terminated UTF-8 string.
- dataSize is an integer related to the size, in bytes, of the internal representation data, as described below.
- pData is either NULL or a pointer to an block of data into which the implementation will write the textual form of the internal representation.

If pData is NULL, then the size, in bytes, of the internal representation data is returned in dataSize. Otherwise, dataSize must be the size of the buffer, in bytes, pointed to by pData and on return dataSize is overwritten with the number of bytes of data actually written to pData including any trailing null character. If dataSize is less than the size, in bytes, of the internal representation data, at most dataSize bytes of data will be written to pData, and VK_INCOMPLETE will be returned instead of VK_SUCCESS, to indicate that not all the available representation was returned.

If isText is VK_TRUE and pData is not NULL and dataSize is not zero, the last byte written to pData will be a null character.

Valid Usage (Implicit)

- VUID-VkPipelineExecutableInternalRepresentationKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INTERNAL_REPRESENTATION_KHR

- VUID-VkPipelineExecutableInternalRepresentationKHR-pNext-pNext
  pNext must be NULL

Information about a particular shader that has been compiled as part of a pipeline object can be extracted by calling:
// Provided by VK_AMD_shader_info

VkResult vkGetShaderInfoAMD(
    VkDevice device,                  
    VkPipeline pipeline,              
    VkShaderStageFlagBits shaderStage,
    VkShaderInfoTypeAMD infoType,     
    size_t* pInfoSize,                
    void* pInfo);

- **device** is the device that created **pipeline**.
- **pipeline** is the target of the query.
- **shaderStage** identifies the particular shader within the pipeline about which information is being queried.
- **infoType** describes what kind of information is being queried.
- **pInfoSize** is a pointer to a value related to the amount of data the query returns, as described below.
- **pInfo** is either **NULL** or a pointer to a buffer.

If **pInfo** is **NULL**, then the maximum size of the information that **can** be retrieved about the shader, in bytes, is returned in **pInfoSize**. Otherwise, **pInfoSize** must point to a variable set by the user to the size of the buffer, in bytes, pointed to by **pInfo**, and on return the variable is overwritten with the amount of data actually written to **pInfo**. If **pInfoSize** is less than the maximum size that **can** be retrieved by the pipeline cache, then at most **pInfoSize** bytes will be written to **pInfo**, and **VK_INCOMPLETE** will be returned, instead of **VK_SUCCESS**, to indicate that not all required of the pipeline cache was returned.

Not all information is available for every shader and implementations may not support all kinds of information for any shader. When a certain type of information is unavailable, the function returns **VK_ERROR_FEATURE_NOT_PRESENT**.

If information is successfully and fully queried, the function will return **VK_SUCCESS**.

For **infoType** **VK_SHADER_INFO_TYPE_STATISTICS_AMD**, a **VkShaderStatisticsInfoAMD** structure will be written to the buffer pointed to by **pInfo**. This structure will be populated with statistics regarding the physical device resources used by that shader along with other miscellaneous information and is described in further detail below.

For **infoType** **VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD**, **pInfo** is a pointer to a UTF-8 null-terminated string containing human-readable disassembly. The exact formatting and contents of the disassembly string are vendor-specific.

The formatting and contents of all other types of information, including **infoType** **VK_SHADER_INFO_TYPE_BINARY_AMD**, are left to the vendor and are not further specified by this extension.
Valid Usage (Implicit)

- VUID-vkGetShaderInfoAMD-device-parameter
  
  `device must be a valid VkDevice handle`

- VUID-vkGetShaderInfoAMD-pipeline-parameter
  
  `pipeline must be a valid VkPipeline handle`

- VUID-vkGetShaderInfoAMD-shaderStage-parameter
  
  `shaderStage must be a valid VkShaderStageFlagBits value`

- VUID-vkGetShaderInfoAMD-infoType-parameter
  
  `infoType must be a valid VkShaderInfoTypeAMD value`

- VUID-vkGetShaderInfoAMD-pInfoSize-parameter
  
  `pInfoSize must be a valid pointer to a size_t value`

- VUID-vkGetShaderInfoAMD-pInfo-parameter
  
  If the value referenced by `pInfoSize` is not 0, and `pInfo` is not NULL, `pInfo must be a valid pointer to an array of pInfoSize bytes`

- VUID-vkGetShaderInfoAMD-pipeline-parent
  
  `pipeline must have been created, allocated, or retrieved from device`

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_FEATURE_NOT_PRESENT
- VK_ERROR_OUT_OF_HOST_MEMORY

Possible values of `vkGetShaderInfoAMD::infoType`, specifying the information being queried from a shader, are:

```c
// Provided by VK_AMD_shader_info
typedef enum VkShaderInfoTypeAMD {
    VK_SHADER_INFO_TYPE_STATISTICS_AMD = 0,
    VK_SHADER_INFO_TYPE_BINARY_AMD = 1,
    VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD = 2,
} VkShaderInfoTypeAMD;
```

- `VK_SHADER_INFO_TYPE_STATISTICS_AMD` specifies that device resources used by a shader will be queried.
- `VK_SHADER_INFO_TYPE_BINARY_AMD` specifies that implementation-specific information will be
queried.

- **VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD** specifies that human-readable disassembly of a shader.

The **VkShaderStatisticsInfoAMD** structure is defined as:

```c
// Provided by VK_AMD_shader_info
typedef struct VkShaderStatisticsInfoAMD {
    VkShaderStageFlags          shaderStageMask;
    VkShaderResourceUsageAMD    resourceUsage;
    uint32_t                    numPhysicalVgprs;
    uint32_t                    numPhysicalSgprs;
    uint32_t                    numAvailableVgprs;
    uint32_t                    numAvailableSgprs;
    uint32_t                    computeWorkGroupSize[3];
} VkShaderStatisticsInfoAMD;
```

- **shaderStageMask** are the combination of logical shader stages contained within this shader.
- **resourceUsage** is a **VkShaderResourceUsageAMD** structure describing internal physical device resources used by this shader.
- **numPhysicalVgprs** is the maximum number of vector instruction general-purpose registers (VGPRs) available to the physical device.
- **numPhysicalSgprs** is the maximum number of scalar instruction general-purpose registers (SGPRs) available to the physical device.
- **numAvailableVgprs** is the maximum limit of VGPRs made available to the shader compiler.
- **numAvailableSgprs** is the maximum limit of SGPRs made available to the shader compiler.
- **computeWorkGroupSize** is the local workgroup size of this shader in { X, Y, Z } dimensions.

Some implementations may merge multiple logical shader stages together in a single shader. In such cases, **shaderStageMask** will contain a bitmask of all of the stages that are active within that shader. Consequently, if specifying those stages as input to **vkGetShaderInfoAMD**, the same output information may be returned for all such shader stage queries.

The number of available VGPRs and SGPRs (**numAvailableVgprs** and **numAvailableSgprs** respectively) are the shader-addressable subset of physical registers that is given as a limit to the compiler for register assignment. These values may further be limited by implementations due to performance optimizations where register pressure is a bottleneck.

The **VkShaderResourceUsageAMD** structure is defined as:
typedef struct VkShaderResourceUsageAMD {
    uint32_t numUsedVgprs;
    uint32_t numUsedSgprs;
    uint32_t ldsSizePerLocalWorkGroup;
    size_t ldsUsageSizeInBytes;
    size_t scratchMemUsageInBytes;
} VkShaderResourceUsageAMD;

- `numUsedVgprs` is the number of vector instruction general-purpose registers used by this shader.
- `numUsedSgprs` is the number of scalar instruction general-purpose registers used by this shader.
- `ldsSizePerLocalWorkGroup` is the maximum local data store size per work group in bytes.
- `ldsUsageSizeInBytes` is the LDS usage size in bytes per work group by this shader.
- `scratchMemUsageInBytes` is the scratch memory usage in bytes by this shader.

10.13. Pipeline Compiler Control

The compilation of a pipeline can be tuned by adding a `VkPipelineCompilerControlCreateInfoAMD` structure to the `pNext` chain of `VkGraphicsPipelineCreateInfo` or `VkComputePipelineCreateInfo`.

typedef struct VkPipelineCompilerControlCreateInfoAMD {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCompilerControlFlagsAMD compilerControlFlags;
} VkPipelineCompilerControlCreateInfoAMD;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `compilerControlFlags` is a bitmask of `VkPipelineCompilerControlFlagBitsAMD` affecting how the pipeline will be compiled.

Valid Usage (Implicit)

- VUID-VkPipelineCompilerControlCreateInfoAMD-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_COMPILER_CONTROL_CREATE_INFO_AMD`

- VUID-VkPipelineCompilerControlCreateInfoAMD-compilerControlFlags-zerobitmask
  `compilerControlFlags` must be `0`

There are currently no available flags for this extension; flags will be added by future versions of this extension.
typedef enum VkPipelineCompilerControlFlagBitsAMD {
} VkPipelineCompilerControlFlagBitsAMD;

typedef VkFlags VkPipelineCompilerControlFlagsAMD;

VkPipelineCompilerControlFlagsAMD is a bitmask type for setting a mask of zero or more VkPipelineCompilerControlFlagBitsAMD.

10.14. Pipeline Creation Feedback

Feedback about the creation of a particular pipeline object can be obtained by adding a VkPipelineCreationFeedbackCreateInfoEXT structure to the pNext chain of VkGraphicsPipelineCreateInfo, VkRayTracingPipelineCreateInfoKHR, VkRayTracingPipelineCreateInfoNV, or VkComputePipelineCreateInfo. The VkPipelineCreationFeedbackCreateInfoEXT structure is defined as:

typedef struct VkPipelineCreationFeedbackCreateInfoEXT {
    VkStructureType                   sType;
    const void*                       pNext;
    VkPipelineCreationFeedbackEXT*    pPipelineCreationFeedback;
    uint32_t                          pipelineStageCreationFeedbackCount;
    VkPipelineCreationFeedbackEXT*    pPipelineStageCreationFeedbacks;
} VkPipelineCreationFeedbackCreateInfoEXT;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• pPipelineCreationFeedback is a pointer to a VkPipelineCreationFeedbackEXT structure.
• pipelineStageCreationFeedbackCount is the number of elements in pPipelineStageCreationFeedbacks.
• pPipelineStageCreationFeedbacks is a pointer to an array of pipelineStageCreationFeedbackCount VkPipelineCreationFeedbackEXT structures.

An implementation should write pipeline creation feedback to pPipelineCreationFeedback and may write pipeline stage creation feedback to pPipelineStageCreationFeedbacks. An implementation must set or clear the VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT in VkPipelineCreationFeedbackEXT::flags for pPipelineCreationFeedback and every element of pPipelineStageCreationFeedbacks.
**Note**

One common scenario for an implementation to skip per-stage feedback is when `VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT` is set in `pPipelineCreationFeedback`.

When chained to `VkRayTracingPipelineCreateInfoKHR`, `VkRayTracingPipelineCreateInfoNV`, or `VkGraphicsPipelineCreateInfo`, the `i` element of `pPipelineStageCreationFeedbacks` corresponds to the `i` element of ` VkRayTracingPipelineCreateInfoKHR::pStages`, ` VkRayTracingPipelineCreateInfoNV::pStages`, or `VkGraphicsPipelineCreateInfo::pStages`. When chained to `VkComputePipelineCreateInfo`, the first element of `pPipelineStageCreationFeedbacks` corresponds to `VkComputePipelineCreateInfo::stage`.

### Valid Usage

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pipelineStageCreationFeedbackCount-02668**
  
  When chained to `VkGraphicsPipelineCreateInfo`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` must equal `VkGraphicsPipelineCreateInfo::stageCount`

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pipelineStageCreationFeedbackCount-02669**
  
  When chained to `VkComputePipelineCreateInfo`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` must equal 1

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pipelineStageCreationFeedbackCount-02670**
  
  When chained to `VkRayTracingPipelineCreateInfoKHR`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` must equal `VkRayTracingPipelineCreateInfoKHR::stageCount`

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pipelineStageCreationFeedbackCount-02969**
  
  When chained to `VkRayTracingPipelineCreateInfoNV`, `VkPipelineCreationFeedbackEXT::pipelineStageCreationFeedbackCount` must equal `VkRayTracingPipelineCreateInfoNV::stageCount`

### Valid Usage (Implicit)

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO_EXT`

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pPipelineCreationFeedback-parameter**
  
  `pPipelineCreationFeedback` must be a valid pointer to a `VkPipelineCreationFeedbackEXT` structure

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pPipelineStageCreationFeedbacks-parameter**
  
  `pPipelineStageCreationFeedbacks` must be a valid pointer to an array of `VkPipelineCreationFeedbackEXT` structures

- **VUID-VkPipelineCreationFeedbackCreateInfoEXT-pipelineStageCreationFeedbackCount-arraylength**
  
  `pipelineStageCreationFeedbackCount` must be greater than 0
The `VkPipelineCreationFeedbackEXT` structure is defined as:

```c
// Provided by VK_EXT_pipeline_creation_feedback
typedef struct VkPipelineCreationFeedbackEXT {
    VkPipelineCreationFeedbackFlagsEXT flags;
    uint64_t duration;
} VkPipelineCreationFeedbackEXT;
```

- `flags` is a bitmask of `VkPipelineCreationFeedbackFlagBitsEXT` providing feedback about the creation of a pipeline or of a pipeline stage.
- `duration` is the duration spent creating a pipeline or pipeline stage in nanoseconds.

If the `VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT` is not set in `flags`, an implementation must not set any other bits in `flags`, and the values of all other `VkPipelineCreationFeedbackEXT` data members are undefined.

Possible values of the `flags` member of `VkPipelineCreationFeedbackEXT` are:

```c
// Provided by VK_EXT_pipeline_creation_feedback
typedef enum VkPipelineCreationFeedbackFlagBitsEXT {
    VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT = 0x00000001,
    VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT = 0x00000002,
    VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT_EXT = 0x00000004,
} VkPipelineCreationFeedbackFlagBitsEXT;
```

- `VK_PIPELINE_CREATION_FEEDBACK_VALID_BIT_EXT` indicates that the feedback information is valid.
- `VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT` indicates that a readily usable pipeline or pipeline stage was found in the `pipelineCache` specified by the application in the pipeline creation command.

An implementation should set the `VK_PIPELINE_CREATION_FEEDBACK_APPLICATION_PIPELINE_CACHE_HIT_BIT_EXT` bit if it was able to avoid the large majority of pipeline or pipeline stage creation work by using the `pipelineCache` parameter of `vkCreateGraphicsPipelines`, `vkCreateRayTracingPipelinesKHR`, `vkCreateRayTracingPipelinesNV`, or `vkCreateComputePipelines`. When an implementation sets this bit for the entire pipeline, it may leave it unset for any stage.

**Note**

Implementations are encouraged to provide a meaningful signal to applications using this bit. The intention is to communicate to the application that the pipeline or pipeline stage was created "as fast as it gets" using the pipeline cache provided by the application. If an implementation uses an internal cache, it is discouraged from setting this bit as the feedback would be unactionable.

- `VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT_EXT` indicates that the base
pipeline specified by the `basePipelineHandle` or `basePipelineIndex` member of the `Vk*PipelineCreateInfo` structure was used to accelerate the creation of the pipeline.

An implementation should set the `VK_PIPELINE_CREATION_FEEDBACK_BASE_PIPELINE_ACCELERATION_BIT_EXT` bit if it was able to avoid a significant amount of work by using the base pipeline.

**Note**

While "significant amount of work" is subjective, implementations are encouraged to provide a meaningful signal to applications using this bit. For example, a 1% reduction in duration may not warrant setting this bit, while a 50% reduction would.

```c
// Provided by VK_EXT_pipeline_creation_feedback
typedef VkFlags VkPipelineCreationFeedbackFlagsEXT;
```

`VkPipelineCreationFeedbackFlagsEXT` is a bitmask type for providing zero or more `VkPipelineCreationFeedbackFlagBitsEXT`. 
Chapter 11. Memory Allocation

Vulkan memory is broken up into two categories, *host memory* and *device memory*.

### 11.1. Host Memory

Host memory is memory needed by the Vulkan implementation for non-device-visible storage.

#### Note

This memory may be used to store the implementation’s representation and state of Vulkan objects.

Vulkan provides applications the opportunity to perform host memory allocations on behalf of the Vulkan implementation. If this feature is not used, the implementation will perform its own memory allocations. Since most memory allocations are off the critical path, this is not meant as a performance feature. Rather, this can be useful for certain embedded systems, for debugging purposes (e.g. putting a guard page after all host allocations), or for memory allocation logging.

Allocators are provided by the application as a pointer to a `VkAllocationCallbacks` structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkAllocationCallbacks {
    void* pUserData;
    PFN_vkAllocationFunction pfnAllocation;
    PFN_vkReallocationFunction pfnReallocation;
    PFN_vkFreeFunction pfnFree;
    PFN_vkInternalAllocationNotification pfnInternalAllocation;
    PFN_vkInternalFreeNotification pfnInternalFree;
} VkAllocationCallbacks;
```

- `pUserData` is a value to be interpreted by the implementation of the callbacks. When any of the callbacks in `VkAllocationCallbacks` are called, the Vulkan implementation will pass this value as the first parameter to the callback. This value can vary each time an allocator is passed into a command, even when the same object takes an allocator in multiple commands.

- `pfnAllocation` is a `PFN_vkAllocationFunction` pointer to an application-defined memory allocation function.

- `pfnReallocation` is a `PFN_vkReallocationFunction` pointer to an application-defined memory reallocation function.

- `pfnFree` is a `PFN_vkFreeFunction` pointer to an application-defined memory free function.

- `pfnInternalAllocation` is a `PFN_vkInternalAllocationNotification` pointer to an application-defined function that is called by the implementation when the implementation makes internal allocations.

- `pfnInternalFree` is a `PFN_vkInternalFreeNotification` pointer to an application-defined function that is called by the implementation when the implementation frees internal allocations.
Valid Usage

- VUID-VkAllocationCallbacks-pfnAllocation-00632
  \textit{pfnAllocation} must be a valid pointer to a valid user-defined \texttt{PFN_vkAllocationFunction}

- VUID-VkAllocationCallbacks-pfnReallocation-00633
  \textit{pfnReallocation} must be a valid pointer to a valid user-defined \texttt{PFN_vkReallocationFunction}

- VUID-VkAllocationCallbacks-pfnFree-00634
  \textit{pfnFree} must be a valid pointer to a valid user-defined \texttt{PFN_vkFreeFunction}

- VUID-VkAllocationCallbacks-pfnInternalAllocation-00635
  If either of \textit{pfnInternalAllocation} or \textit{pfnInternalFree} is not NULL, both must be valid callbacks

The type of \textit{pfnAllocation} is:

\begin{verbatim}
// Provided by VK_VERSION_1_0
typedef void* (VKAPI_PTR *PFN_vkAllocationFunction)(
    void* pUserData,
    size_t size,
    size_t alignment,
    VkSystemAllocationScope allocationScope);
\end{verbatim}

- \texttt{pUserData} is the value specified for \texttt{VkAllocationCallbacks::pUserData} in the allocator specified by the application.
- \texttt{size} is the size in bytes of the requested allocation.
- \texttt{alignment} is the requested alignment of the allocation in bytes and must be a power of two.
- \texttt{allocationScope} is a \texttt{VkSystemAllocationScope} value specifying the allocation scope of the lifetime of the allocation, as described here.

If \textit{pfnAllocation} is unable to allocate the requested memory, it must return \texttt{NULL}. If the allocation was successful, it must return a valid pointer to memory allocation containing at least \texttt{size} bytes, and with the pointer value being a multiple of \texttt{alignment}.

\textbf{Note}

Correct Vulkan operation cannot be assumed if the application does not follow these rules.

For example, \textit{pfnAllocation} (or \textit{pfnReallocation}) could cause termination of running Vulkan instance(s) on a failed allocation for debugging purposes, either directly or indirectly. In these circumstances, it cannot be assumed that any part of any affected \texttt{VkInstance} objects are going to operate correctly (even \texttt{vkDestroyInstance}), and the application must ensure it cleans up properly via other means (e.g. process termination).
If pfnAllocation returns NULL, and if the implementation is unable to continue correct processing of the current command without the requested allocation, it must treat this as a runtime error, and generate VK_ERROR_OUT_OF_HOST_MEMORY at the appropriate time for the command in which the condition was detected, as described in Return Codes.

If the implementation is able to continue correct processing of the current command without the requested allocation, then it may do so, and must not generate VK_ERROR_OUT_OF_HOST_MEMORY as a result of this failed allocation.

The type of pfnReallocation is:

```c
// Provided by VK_VERSION_1_0
typedef void* (VKAPI_PTR *PFN_vkReallocationFunction)(
    void* pData,
    void* pOriginal,
    size_t size,
    size_t alignment,
    VkSystemAllocationScope allocationScope);
```

- pData is the value specified for VkAllocationCallbacks::pUserData in the allocator specified by the application.
- pOriginal must be either NULL or a pointer previously returned by pfnReallocation or pfnAllocation of a compatible allocator.
- size is the size in bytes of the requested allocation.
- alignment is the requested alignment of the allocation in bytes and must be a power of two.
- allocationScope is a VkSystemAllocationScope value specifying the allocation scope of the lifetime of the allocation, as described here.

pfnReallocation must return an allocation with enough space for size bytes, and the contents of the original allocation from bytes zero to \text{min}(original size, new size) - 1 must be preserved in the returned allocation. If size is larger than the old size, the contents of the additional space are undefined. If satisfying these requirements involves creating a new allocation, then the old allocation should be freed.

If pOriginal is NULL, then pfnReallocation must behave equivalently to a call to PFN_vkAllocationFunction with the same parameter values (without pOriginal).

If size is zero, then pfnReallocation must behave equivalently to a call to PFN_vkFreeFunction with the same pData parameter value, and pMemory equal to pOriginal.

If pOriginal is non-NULL, the implementation must ensure that alignment is equal to the alignment used to originally allocate pOriginal.

If this function fails and pOriginal is non-NULL the application must not free the old allocation.

pfnReallocation must follow the same rules for return values as PFN_vkAllocationFunction.

The type of pfnFree is:
**typedef void (VKAPI_PTR *PFN_vkFreeFunction)(
    void* pUserData,
    void* pMemory);**

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `pMemory` is the allocation to be freed.

`pMemory` **may** be `NULL`, which the callback **must** handle safely. If `pMemory` is non-`NULL`, it **must** be a pointer previously allocated by `pfnAllocation` or `pfnReallocation`. The application **should** free this memory.

The type of `PFN_vkFreeFunction` is:

**typedef void (VKAPI_PTR *PFN_vkInternalAllocationNotification)(
    void* pUserData,
    size_t size,
    VkInternalAllocationType allocationType,
    VkSystemAllocationScope allocationScope);**

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `size` is the requested size of an allocation.
- `allocationType` is a `VkInternalAllocationType` value specifying the requested type of an allocation.
- `allocationScope` is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described [here](#).

This is a purely informational callback.

The type of `PFN_vkInternalAllocationNotification` is:

**typedef void (VKAPI_PTR *PFN_vkInternalFreeNotification)(
    void* pUserData,
    size_t size,
    VkInternalAllocationType allocationType,
    VkSystemAllocationScope allocationScope);**

- `pUserData` is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- `size` is the requested size of an allocation.
• allocationType is a `VkInternalAllocationType` value specifying the requested type of an allocation.

• allocationScope is a `VkSystemAllocationScope` value specifying the allocation scope of the lifetime of the allocation, as described here.

Each allocation has an allocation scope defining its lifetime and which object it is associated with. Possible values passed to the allocationScope parameter of the callback functions specified by `VkAllocationCallbacks`, indicating the allocation scope, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSystemAllocationScope {
    VK_SYSTEM_ALLOCATION_SCOPE_COMMAND = 0,
    VK_SYSTEM_ALLOCATION_SCOPE_OBJECT = 1,
    VK_SYSTEM_ALLOCATION_SCOPE_CACHE = 2,
    VK_SYSTEM_ALLOCATION_SCOPE_DEVICE = 3,
    VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE = 4,
} VkSystemAllocationScope;
```

• `VK_SYSTEM_ALLOCATION_SCOPE_COMMAND` specifies that the allocation is scoped to the duration of the Vulkan command.

• `VK_SYSTEM_ALLOCATION_SCOPE_OBJECT` specifies that the allocation is scoped to the lifetime of the Vulkan object that is being created or used.

• `VK_SYSTEM_ALLOCATION_SCOPE_CACHE` specifies that the allocation is scoped to the lifetime of a `VkPipelineCache` or `VkValidationCacheEXT` object.

• `VK_SYSTEM_ALLOCATION_SCOPE_DEVICE` specifies that the allocation is scoped to the lifetime of the Vulkan device.

• `VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE` specifies that the allocation is scoped to the lifetime of the Vulkan instance.

Most Vulkan commands operate on a single object, or there is a sole object that is being created or manipulated. When an allocation uses an allocation scope of `VK_SYSTEM_ALLOCATION_SCOPE_OBJECT` or `VK_SYSTEM_ALLOCATION_SCOPE_CACHE`, the allocation is scoped to the object being created or manipulated.

When an implementation requires host memory, it will make callbacks to the application using the most specific allocator and allocation scope available:

• If an allocation is scoped to the duration of a command, the allocator will use the `VK_SYSTEM_ALLOCATION_SCOPE_COMMAND` allocation scope. The most specific allocator available is used: if the object being created or manipulated has an allocator, that object's allocator will be used, else if the parent `VkDevice` has an allocator it will be used, else if the parent `VkInstance` has an allocator it will be used. Else,

• If an allocation is associated with a `VkValidationCacheEXT` or `VkPipelineCache` object, the allocator will use the `VK_SYSTEM_ALLOCATION_SCOPE_CACHE` allocation scope. The most specific allocator available is used (cache, else device, else instance). Else,
• If an allocation is scoped to the lifetime of an object, that object is being created or manipulated by the command, and that object's type is not VkDevice or VkInstance, the allocator will use an allocation scope of VK_SYSTEM_ALLOCATION_SCOPE_OBJECT. The most specific allocator available is used (object, else device, else instance). Else,

• If an allocation is scoped to the lifetime of a device, the allocator will use an allocation scope of VK_SYSTEM_ALLOCATION_SCOPE_DEVICE. The most specific allocator available is used (device, else instance). Else,

• If the allocation is scoped to the lifetime of an instance and the instance has an allocator, its allocator will be used with an allocation scope of VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE.

• Otherwise an implementation will allocate memory through an alternative mechanism that is unspecified.

Objects that are allocated from pools do not specify their own allocator. When an implementation requires host memory for such an object, that memory is sourced from the object's parent pool's allocator.

The application is not expected to handle allocating memory that is intended for execution by the host due to the complexities of differing security implementations across multiple platforms. The implementation will allocate such memory internally and invoke an application provided informational callback when these internal allocations are allocated and freed. Upon allocation of executable memory, pfnInternalAllocation will be called. Upon freeing executable memory, pfnInternalFree will be called. An implementation will only call an informational callback for executable memory allocations and frees.

The allocationType parameter to the pfnInternalAllocation and pfnInternalFree functions may be one of the following values:

```c
// Provided by VK_VERSION_1_0
typedef enum VkInternalAllocationType {
    VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE = 0,
} VkInternalAllocationType;
```

• VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE specifies that the allocation is intended for execution by the host.

An implementation must only make calls into an application-provided allocator during the execution of an API command. An implementation must only make calls into an application-provided allocator from the same thread that called the provoking API command. The implementation should not synchronize calls to any of the callbacks. If synchronization is needed, the callbacks must provide it themselves. The informational callbacks are subject to the same restrictions as the allocation callbacks.

If an implementation intends to make calls through a VkAllocationCallbacks structure between the time a vkCreate* command returns and the time a corresponding vkDestroy* command begins, that implementation must save a copy of the allocator before the vkCreate* command returns. The callback functions and any data structures they rely upon must remain valid for the lifetime of the object they are associated with.
If an allocator is provided to a `vkCreate*` command, a compatible allocator **must** be provided to the corresponding `vkDestroy*` command. Two `VkAllocationCallbacks` structures are compatible if memory allocated with `pfnAllocation` or `pfnReallocation` in each **can** be freed with `pfnReallocation` or `pfnFree` in the other. An allocator **must** not be provided to a `vkDestroy*` command if an allocator was not provided to the corresponding `vkCreate*` command.

If a non-NULL allocator is used, the `pfnAllocation`, `pfnReallocation` and `pfnFree` members **must** be non-NULL and point to valid implementations of the callbacks. An application **can** choose to not provide informational callbacks by setting both `pfnInternalAllocation` and `pfnInternalFree` to `NULL`. `pfnInternalAllocation` and `pfnInternalFree` **must** either both be `NULL` or both be non-NULL.

If `pfnAllocation` or `pfnReallocation` fail, the implementation **may** fail object creation and/or generate a `VK_ERROR_OUT_OF_HOST_MEMORY` error, as appropriate.

Allocation callbacks **must** not call any Vulkan commands.

The following sets of rules define when an implementation is permitted to call the allocator callbacks.

`pfnAllocation` or `pfnReallocation` **may** be called in the following situations:

- Allocations scoped to a `VkDevice` or `VkInstance` **may** be allocated from any API command.
- Allocations scoped to a command **may** be allocated from any API command.
- Allocations scoped to a `VkPipelineCache` **may** only be allocated from:
  - `vkCreatePipelineCache`
  - `vkMergePipelineCaches` for `dstCache`
  - `vkCreateGraphicsPipelines` for `pipelineCache`
  - `vkCreateComputePipelines` for `pipelineCache`
- Allocations scoped to a `VkValidationCacheEXT` **may** only be allocated from:
  - `vkCreateValidationCacheEXT`
  - `vkMergeValidationCachesEXT` for `dstCache`
  - `vkCreateShaderModule` for `validationCache` in `VkShaderModuleValidationCacheCreateInfoEXT`
- Allocations scoped to a `VkDescriptorPool` **may** only be allocated from:
  - any command that takes the pool as a direct argument
  - `vkAllocateDescriptorSets` for the `descriptorPool` member of its `pAllocateInfo` parameter
  - `vkCreateDescriptorPool`
- Allocations scoped to a `VkCommandPool` **may** only be allocated from:
  - any command that takes the pool as a direct argument
  - `vkCreateCommandPool`
  - `vkAllocateCommandBuffers` for the `commandPool` member of its `pAllocateInfo` parameter
  - any `vkCmd*` command whose `commandBuffer` was allocated from that `VkCommandPool`
• Allocations scoped to any other object may only be allocated in that object's `vkCreate*` command.

`PFNFree`, or `PFNReallocation` with zero size, may be called in the following situations:

• Allocations scoped to a `VkDevice` or `VkInstance` may be freed from any API command.
• Allocations scoped to a command must be freed by any API command which allocates such memory.
• Allocations scoped to a `VkPipelineCache` may be freed from `vkDestroyPipelineCache`.
• Allocations scoped to a `VkValidationCacheEXT` may be freed from `vkDestroyValidationCacheEXT`.
• Allocations scoped to a `VkDescriptorPool` may be freed from
  ◦ any command that takes the pool as a direct argument
• Allocations scoped to a `VkCommandPool` may be freed from:
  ◦ any command that takes the pool as a direct argument
  ◦ `vkResetCommandBuffer` whose `commandBuffer` was allocated from that `VkCommandPool`
• Allocations scoped to any other object may be freed in that object's `vkDestroy*` command.
• Any command that allocates host memory may also free host memory of the same scope.

### 11.2. Device Memory

Device memory is memory that is visible to the device — for example the contents of the image or buffer objects, which can be natively used by the device.

Memory properties of a physical device describe the memory heaps and memory types available.

To query memory properties, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceMemoryProperties(
    VkPhysicalDevice                            physicalDevice,
    VkPhysicalDeviceMemoryProperties*           pMemoryProperties);
```

• `physicalDevice` is the handle to the device to query.
• `pMemoryProperties` is a pointer to a `VkPhysicalDeviceMemoryProperties` structure in which the properties are returned.
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceMemoryProperties-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceMemoryProperties-pMemoryProperties-parameter
  pMemoryProperties must be a valid pointer to a VkPhysicalDeviceMemoryProperties structure

The VkPhysicalDeviceMemoryProperties structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceMemoryProperties {
    uint32_t memoryTypeCount;
    VkMemoryType memoryTypes[VK_MAX_MEMORY_TYPES];
    uint32_t memoryHeapCount;
    VkMemoryHeap memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;
```

- memoryTypeCount is the number of valid elements in the memoryTypes array.
- memoryTypes is an array of VK_MAX_MEMORY_TYPES VkMemoryType structures describing the memory types that can be used to access memory allocated from the heaps specified by memoryHeaps.
- memoryHeapCount is the number of valid elements in the memoryHeaps array.
- memoryHeaps is an array of VK_MAX_MEMORY_HEAPS VkMemoryHeap structures describing the memory heaps from which memory can be allocated.

The VkPhysicalDeviceMemoryProperties structure describes a number of memory heaps as well as a number of memory types that can be used to access memory allocated in those heaps. Each heap describes a memory resource of a particular size, and each memory type describes a set of memory properties (e.g. host cached vs uncached) that can be used with a given memory heap. Allocations using a particular memory type will consume resources from the heap indicated by that memory type’s heap index. More than one memory type may share each heap, and the heaps and memory types provide a mechanism to advertise an accurate size of the physical memory resources while allowing the memory to be used with a variety of different properties.

The number of memory heaps is given by memoryHeapCount and is less than or equal to VK_MAX_MEMORY_HEAPS. Each heap is described by an element of the memoryHeaps array as a VkMemoryHeap structure. The number of memory types available across all memory heaps is given by memoryTypeCount and is less than or equal to VK_MAX_MEMORY_TYPES. Each memory type is described by an element of the memoryTypes array as a VkMemoryType structure.

At least one heap must include VK_MEMORY_HEAP_DEVICE_LOCAL_BIT in VkMemoryHeap::flags. If there are multiple heaps that all have similar performance characteristics, they may all include VK_MEMORY_HEAP_DEVICE_LOCAL_BIT. In a unified memory architecture (UMA) system there is often only a single memory heap which is considered to be equally “local” to the host and to the device,
and such an implementation **must** advertise the heap as device-local.

Each memory type returned by `vkGetPhysicalDeviceMemoryProperties` **must** have its `propertyFlags` set to one of the following values:

- 0
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT`
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT`
- `VK_MEMORYPROPERTYDEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORYPROPERTY_HOST_CACHED_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORYPROPERTY_HOST_CACHED_BIT` | `VK_MEMORYPROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORYPROPERTYDEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTYHOST_VISIBLE_BIT` | `VK_MEMORYPROPERTYHOST_CACHED_BIT` | `VK_MEMORYPROPERTYHOST_COHERENT_BIT` | `VK_MEMORYPROPERTYDEVICE_COHERENT_BIT_AMD`
- `VK_MEMORY_PROPERTY_PROTECTED_BIT`
- `VK_MEMORY_PROPERTY_PROTECTED_BIT` | `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`
- `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORYPROPERTY_HOST_COHERENT_BIT` | `VK_MEMORYPROPERTYDEVICE_COHERENT_BIT_AMD`
- `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORYPROPERTY_HOST_CACHED_BIT` | `VK_MEMORYPROPERTY_HOST_COHERENT_BIT` | `VK_MEMORYPROPERTYDEVICE_COHERENT_BIT_AMD`
- `VK_MEMORYPROPERTYDEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTYDEVICE_COHERENT_BIT_AMD`
- `VK_MEMORYPROPERTYDEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORYPROPERTY_HOST_COHERENT_BIT` | `VK_MEMORYPROPERTYDEVICE_COHERENT_BIT_AMD`
- `VK_MEMORYPROPERTYDEVICE_LOCAL_BIT` | `VK_MEMORYPROPERTYDEVICE_COHERENT_BIT_AMD`
There must be at least one memory type with both the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` bits set in its `propertyFlags`. There must be at least one memory type with the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` bit set in its `propertyFlags`. If the `deviceCoherentMemory` feature is enabled, there must be at least one memory type with the `VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD` bit set in its `propertyFlags`.

For each pair of elements \(X\) and \(Y\) returned in `memoryTypes`, \(X\) must be placed at a lower index position than \(Y\) if:

- the set of bit flags returned in the `propertyFlags` member of \(X\) is a strict subset of the set of bit flags returned in the `propertyFlags` member of \(Y\); or
- the `propertyFlags` members of \(X\) and \(Y\) are equal, and \(X\) belongs to a memory heap with greater performance (as determined in an implementation-specific manner); or
- the `propertyFlags` members of \(Y\) includes `VK_MEMORY_PROPERTYDEVICE_COHERENT_BIT_AMD` or `VK_MEMORY_PROPERTYDEVICE_UNCACHED_BIT_AMD` and \(X\) does not
**Note**

There is no ordering requirement between X and Y elements for the case their `propertyFlags` members are not in a subset relation. That potentially allows more than one possible way to order the same set of memory types. Notice that the list of all allowed memory property flag combinations is written in a valid order. But if instead `VK_MEMORYPROPERTYDEVICELOCALBIT` was before `VK_MEMORYPROPERTYHOSTVISIBLEBIT | VK_MEMORYPROPERTYHOSTCOHERENTBIT`, the list would still be in a valid order.

There may be a performance penalty for using device coherent or uncached device memory types, and using these accidentally is undesirable. In order to avoid this, memory types with these properties always appear at the end of the list; but are subject to the same rules otherwise.

This ordering requirement enables applications to use a simple search loop to select the desired memory type along the lines of:
Find a memory in `memoryTypeBitsRequirement` that includes all of `requiredProperties`:

```c
int32_t findProperties(const VkPhysicalDeviceMemoryProperties* pMemoryProperties,
                        uint32_t memoryTypeBitsRequirement,
                        VkMemoryPropertyFlags requiredProperties) {
  const uint32_t memoryCount = pMemoryProperties->memoryTypeCount;
  for (uint32_t memoryIndex = 0; memoryIndex < memoryCount; ++memoryIndex) {
    const uint32_t memoryTypeBits = (1 << memoryIndex);
    const bool isRequiredMemoryType = memoryTypeBitsRequirement & memoryTypeBits;
    const VkMemoryPropertyFlags properties = pMemoryProperties->memoryTypes[memoryIndex].propertyFlags;
    const bool hasRequiredProperties = (properties & requiredProperties) == requiredProperties;
    if (isRequiredMemoryType && hasRequiredProperties)
      return static_cast<int32_t>(memoryIndex);
  }

  // failed to find memory type
  return -1;
}
```

Try to find an optimal memory type, or if it does not exist try fallback memory type:

- `device` is the VkDevice
- `image` is the VkImage that requires memory to be bound
- `memoryProperties` properties as returned by `vkGetPhysicalDeviceMemoryProperties`
- `requiredProperties` are the property flags that must be present
- `optimalProperties` are the property flags that are preferred by the application

```c
VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(device, image, &memoryRequirements);
int32_t memoryType = findProperties(&memoryProperties, memoryRequirements.memoryTypeBits,
                                      optimalProperties);
if (memoryType == -1) // not found; try fallback properties
  memoryType = findProperties(&memoryProperties, memoryRequirements.memoryTypeBits,
                               requiredProperties);
```

To query memory properties, call:

```c
void vkGetPhysicalDeviceMemoryProperties2(VkPhysicalDevice physicalDevice,
                                          VkPhysicalDeviceMemoryProperties2* pMemoryProperties);
```

or the equivalent command
```c
void vkGetPhysicalDeviceMemoryProperties2KHR(  
    VkPhysicalDevice                            physicalDevice,  
    VkPhysicalDeviceMemoryProperties2*          pMemoryProperties);
```

- `physicalDevice` is the handle to the device to query.
- `pMemoryProperties` is a pointer to a `VkPhysicalDeviceMemoryProperties2` structure in which the properties are returned.

`vkGetPhysicalDeviceMemoryProperties2` behaves similarly to `vkGetPhysicalDeviceMemoryProperties`, with the ability to return extended information in a `pNext` chain of output structures.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceMemoryProperties2-physicalDevice-parameter  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceMemoryProperties2-pMemoryProperties-parameter  
  `pMemoryProperties` must be a valid pointer to a `VkPhysicalDeviceMemoryProperties2` structure

The `VkPhysicalDeviceMemoryProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1  
typedef struct VkPhysicalDeviceMemoryProperties2 {  
    VkStructureType                     sType;  
    void*                               pNext;  
    VkPhysicalDeviceMemoryProperties    memoryProperties;  
} VkPhysicalDeviceMemoryProperties2;
```

or the equivalent

```c
// Provided by VK_KHR_get_physical_device_properties2
typedef VkPhysicalDeviceMemoryProperties2 VkPhysicalDeviceMemoryProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memoryProperties` is a `VkPhysicalDeviceMemoryProperties` structure which is populated with the same values as in `vkGetPhysicalDeviceMemoryProperties`.

648 | Chapter 11. Memory Allocation
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMemoryProperties2-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2

- VUID-VkPhysicalDeviceMemoryProperties2-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of
  VkPhysicalDeviceMemoryBudgetPropertiesEXT

- VUID-VkPhysicalDeviceMemoryProperties2-sType-unique
  The sType value of each struct in the pNext chain must be unique

The VkMemoryHeap structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryHeap {
    VkDeviceSize size;
    VkMemoryHeapFlags flags;
} VkMemoryHeap;
```

- size is the total memory size in bytes in the heap.
- flags is a bitmask of VkMemoryHeapFlagBits specifying attribute flags for the heap.

Bits which may be set in VkMemoryHeap::flags, indicating attribute flags for the heap, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkMemoryHeapFlagBits {
    VK_MEMORY_HEAP_DEVICE_LOCAL_BIT = 0x00000001,
    // Provided by VK_VERSION_1_1
    VK_MEMORY_HEAP_MULTI_INSTANCE_BIT = 0x00000002,
    // Provided by VK_KHR_device_group_creation
    VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHR = VK_MEMORY_HEAP_MULTI_INSTANCE_BIT,
} VkMemoryHeapFlagBits;
```

- VK_MEMORY_HEAP_DEVICE_LOCAL_BIT specifies that the heap corresponds to device local memory. Device local memory may have different performance characteristics than host local memory, and may support different memory property flags.
- VK_MEMORY_HEAP_MULTI_INSTANCE_BIT specifies that in a logical device representing more than one physical device, there is a per-physical device instance of the heap memory. By default, an allocation from such a heap will be replicated to each physical device's instance of the heap.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkMemoryHeapFlags;
```

VkMemoryHeapFlags is a bitmask type for setting a mask of zero or more VkMemoryHeapFlagBits.
The `VkMemoryType` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryType {
    VkMemoryPropertyFlags propertyFlags;
    uint32_t heapIndex;
} VkMemoryType;
```

- `heapIndex` describes which memory heap this memory type corresponds to, and **must** be less than `memoryHeapCount` from the `VkPhysicalDeviceMemoryProperties` structure.
- `propertyFlags` is a bitmask of `VkMemoryPropertyFlagBits` of properties for this memory type.

Bits which **may** be set in `VkMemoryType::propertyFlags`, indicating properties of a memory heap, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkMemoryPropertyFlagBits {
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002,
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
    VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008,
    VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010,
    // Provided by VK_VERSION_1_1
    VK_MEMORY_PROPERTY_PROTECTED_BIT = 0x00000020,
    // Provided by VK_AMD_device_coherent_memory
    VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD = 0x00000040,
    // Provided by VK_AMD_device_coherent_memory
    VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD = 0x00000080,
} VkMemoryPropertyFlagBits;
```

- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` bit specifies that memory allocated with this type is the most efficient for device access. This property will be set if and only if the memory type belongs to a heap with the `VK_MEMORY_HEAP_DEVICE_LOCAL_BIT` set.
- `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` bit specifies that memory allocated with this type **can** be mapped for host access using `vkMapMemory`.
- `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` bit specifies that the host cache management commands `vkFlushMappedMemoryRanges` and `vkInvalidateMappedMemoryRanges` are not needed to flush host writes to the device or make device writes visible to the host, respectively.
- `VK_MEMORY_PROPERTY_HOST_CACHED_BIT` bit specifies that memory allocated with this type is cached on the host. Host memory accesses to uncached memory are slower than to cached memory, however uncached memory is always host coherent.
- `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit specifies that the memory type only allows device access to the memory. Memory types **must** not have both `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` and `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` set. Additionally, the object’s backing memory **may** be provided by the implementation lazily as
specified in *Lazily Allocated Memory*.

- **VK_MEMORYPROPERTY_PROTECTED_BIT** bit specifies that the memory type only allows device access to the memory, and allows protected queue operations to access the memory. Memory types **must not have** **VK_MEMORYPROPERTY_PROTECTED_BIT** set and any of **VK_MEMORYPROPERTY_HOSTVISIBLE_BIT** set, or **VK_MEMORYPROPERTYHOSTCOHERENT_BIT** set, or **VK_MEMORYPROPERTYHOSTCACHED_BIT** set.

- **VK_MEMORYPROPERTYDEVICECOHERENT_BIT_AMD** bit specifies that device accesses to allocations of this memory type are automatically made available and visible.

- **VK_MEMORYPROPERTYDEVICEUNCACHED_BIT_AMD** bit specifies that memory allocated with this type is not cached on the device. Uncached device memory is always device coherent.

For any memory allocated with both the **VK_MEMORYPROPERTYHOSTCOHERENT_BIT** and the **VK_MEMORYPROPERTYDEVICECOHERENT_BIT_AMD**, host or device accesses also perform automatic memory domain transfer operations, such that writes are always automatically available and visible to both host and device memory domains.

**Note**

Device coherence is a useful property for certain debugging use cases (e.g. crash analysis, where performing separate coherence actions could mean values are not reported correctly). However, device coherent accesses may be slower than equivalent accesses without device coherence, particularly if they are also device uncached. For device uncached memory in particular, repeated accesses to the same or neighbouring memory locations over a short time period (e.g. within a frame) may be slower than it would be for the equivalent cached memory type. As such, it is generally inadvisable to use device coherent or device uncached memory except when really needed.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkMemoryPropertyFlags;
```

**VkMemoryPropertyFlags** is a bitmask type for setting a mask of zero or more **VkMemoryPropertyFlagBits**.

If the **VkPhysicalDeviceMemoryBudgetPropertiesEXT** structure is included in the **pNext** chain of **VkPhysicalDeviceMemoryProperties2**, it is filled with the current memory budgets and usages.

The **VkPhysicalDeviceMemoryBudgetPropertiesEXT** structure is defined as:

```c
// Provided by VK_EXT_memory_budget
typedef struct VkPhysicalDeviceMemoryBudgetPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkDeviceSize heapBudget[VK_MAX_MEMORY_HEAPS];
    VkDeviceSize heapUsage[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryBudgetPropertiesEXT;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **heapBudget** is an array of **VK_MAX_MEMORY_HEAPS** **VkDeviceSize** values in which memory budgets are returned, with one element for each memory heap. A heap’s budget is a rough estimate of how much memory the process can allocate from that heap before allocations may fail or cause performance degradation. The budget includes any currently allocated device memory.

• **heapUsage** is an array of **VK_MAX_MEMORY_HEAPS** **VkDeviceSize** values in which memory usages are returned, with one element for each memory heap. A heap’s usage is an estimate of how much memory the process is currently using in that heap.

The values returned in this structure are not invariant. The **heapBudget** and **heapUsage** values must be zero for array elements greater than or equal to **VkPhysicalDeviceMemoryProperties::memoryHeapCount**. The **heapBudget** value must be non-zero for array elements less than **VkPhysicalDeviceMemoryProperties::memoryHeapCount**. The **heapBudget** value must be less than or equal to **VkMemoryHeap::size** for each heap.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMemoryBudgetPropertiesEXT-sType-sType

  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT**

A Vulkan device operates on data in device memory via memory objects that are represented in the API by a **VkDeviceMemory** handle:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDeviceMemory)
```

To allocate memory objects, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkAllocateMemory(
    VkDevice device,
    const VkMemoryAllocateInfo* pAllocateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDeviceMemory* pMemory);
```

- **device** is the logical device that owns the memory.

- **pAllocateInfo** is a pointer to a **VkMemoryAllocateInfo** structure describing parameters of the allocation. A successful returned allocation must use the requested parameters—no substitution is permitted by the implementation.

- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

- **pMemory** is a pointer to a **VkDeviceMemory** handle in which information about the allocated memory is returned.
Allocations returned by `vkAllocateMemory` are guaranteed to meet any alignment requirement of the implementation. For example, if an implementation requires 128 byte alignment for images and 64 byte alignment for buffers, the device memory returned through this mechanism would be 128-byte aligned. This ensures that applications can correctly suballocate objects of different types (with potentially different alignment requirements) in the same memory object.

When memory is allocated, its contents are undefined with the following constraint:

- The contents of unprotected memory must not be a function of data protected memory objects, even if those memory objects were previously freed.

The contents of memory allocated by one application should not be a function of data from protected memory objects of another application, even if those memory objects were previously freed.

The maximum number of valid memory allocations that can exist simultaneously within a `VkDevice` may be restricted by implementation- or platform-dependent limits. The `maxMemoryAllocationCount` feature describes the number of allocations that can exist simultaneously before encountering these internal limits.

For historical reasons, if `maxMemoryAllocationCount` is exceeded, some implementations may return `VK_ERROR_TOO_MANY_OBJECTS`. Exceeding this limit will result in undefined behavior, and an application should not rely on the use of the returned error code in order to identify when the limit is reached.

Some platforms may have a limit on the maximum size of a single allocation. For example, certain systems may fail to create allocations with a size greater than or equal to 4GB. Such a limit is implementation-dependent, and if such a failure occurs then the error `VK_ERROR_OUT_OF_DEVICE_MEMORY` must be returned. This limit is advertised in `VkPhysicalDeviceMaintenance3Properties::maxMemoryAllocationSize`.

The cumulative memory size allocated to a heap can be limited by the size of the specified heap. In such cases, allocated memory is tracked on a per-device and per-heap basis. Some platforms allow overallocation into other heaps. The overallocation behavior can be specified through the `VK_AMD_memory_overallocation_behavior` extension.
Valid Usage

- **VUID-vkAllocateMemory-pAllocateInfo-01713**
  \[
  \text{pAllocateInfo->allocationSize} \text{ must be less than or equal toVkPhysicalDeviceMemoryProperties::memoryHeaps[memindex].size where memindex = VkPhysicalDeviceMemoryProperties::memoryTypes[pAllocateInfo->memoryTypeIndex].heapIndex as returned by } \text{vkGetPhysicalDeviceMemoryProperties for the } \text{VkPhysicalDevice that } \text{device} \text{ was created from.}
  \]

- **VUID-vkAllocateMemory-pAllocateInfo-01714**
  \[
  \text{pAllocateInfo->memoryTypeIndex must be less than VkPhysicalDeviceMemoryProperties::memoryTypeCount as returned by } \text{vkGetPhysicalDeviceMemoryProperties for the } \text{VkPhysicalDevice that } \text{device} \text{ was created from.}
  \]

- **VUID-vkAllocateMemory-deviceCoherentMemory-02790**
  If the \text{deviceCoherentMemory} feature is not enabled, \text{pAllocateInfo->memoryTypeIndex must not identify a memory type supporting VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD.}

- **VUID-vkAllocateMemory-maxMemoryAllocationCount-04101**
  There must be less than VkPhysicalDeviceLimits::maxMemoryAllocationCount device memory allocations currently allocated on the device.

Valid Usage (Implicit)

- **VUID-vkAllocateMemory-device-parameter**
  \text{device must be a valid VkDevice handle}

- **VUID-vkAllocateMemory-pAllocateInfo-parameter**
  \text{pAllocateInfo must be a valid pointer to a valid VkMemoryAllocateInfo structure}

- **VUID-vkAllocateMemory-pAllocator-parameter**
  If \text{pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure}

- **VUID-vkAllocateMemory-pMemory-parameter**
  \text{pMemory must be a valid pointer to a VkDeviceMemory handle}
Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE
- VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR

The `VkMemoryAllocateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryAllocateInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceSize       allocationSize;
    uint32_t           memoryTypeIndex;
} VkMemoryAllocateInfo;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `allocationSize` is the size of the allocation in bytes.
- `memoryTypeIndex` is an index identifying a memory type from the `memoryTypes` array of the `VkPhysicalDeviceMemoryProperties` structure.

The internal data of an allocated device memory object must include a reference to implementation-specific resources, referred to as the memory object's payload. Applications can also import and export that internal data to and from device memory objects to share data between Vulkan instances and other compatible APIs. A `VkMemoryAllocateInfo` structure defines a memory import operation if its `pNext` chain includes one of the following structures:

- `VkImportMemoryWin32HandleInfoKHR` with a non-zero `handleType` value
- `VkImportMemoryFdInfoKHR` with a non-zero `handleType` value
- `VkImportMemoryHostPointerInfoEXT` with a non-zero `handleType` value
- `VkImportAndroidHardwareBufferInfoANDROID` with a non-NULL `buffer` value
- `VkImportMemoryZirconHandleInfoFUCHSIA` with a non-zero `handleType` value

If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, `allocationSize` is ignored. The implementation
must query the size of these allocations from the OS.

Whether device memory objects constructed via a memory import operation hold a reference to their payload depends on the properties of the handle type used to perform the import, as defined below for each valid handle type. Importing memory must not modify the content of the memory. Implementations must ensure that importing memory does not enable the importing Vulkan instance to access any memory or resources in other Vulkan instances other than that corresponding to the memory object imported. Implementations must also ensure accessing imported memory which has not been initialized does not allow the importing Vulkan instance to obtain data from the exporting Vulkan instance or vice-versa.

Note

How exported and imported memory is isolated is left to the implementation, but applications should be aware that such isolation may prevent implementations from placing multiple exportable memory objects in the same physical or virtual page. Hence, applications should avoid creating many small external memory objects whenever possible.

Importing memory must not increase overall heap usage within a system. However, they must affect the following per-process values:

* VkPhysicalDeviceMaintenance3Properties::maxMemoryAllocationCount
* VkPhysicalDeviceMemoryBudgetPropertiesEXT::heapUsage

When performing a memory import operation, it is the responsibility of the application to ensure the external handles and their associated payloads meet all valid usage requirements. However, implementations must perform sufficient validation of external handles and payloads to ensure that the operation results in a valid memory object which will not cause program termination, device loss, queue stalls, or corruption of other resources when used as allowed according to its allocation parameters. If the external handle provided does not meet these requirements, the implementation must fail the memory import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE.
Valid Usage

- **VUID-VkMemoryAllocateInfo-pNext-00639**
  
  If the pNext chain includes a VkExportMemoryAllocateInfo structure, and any of the handle types specified in VkExportMemoryAllocateInfo::handleTypes require a dedicated allocation, as reported by vkGetPhysicalDeviceImageFormatProperties2 in VkExternalImageFormatProperties::externalMemoryProperties.externalMemoryFeatures or VkExternalBufferProperties::externalMemoryProperties.externalMemoryFeatures, the pNext chain must include a VkMemoryDedicatedAllocateInfo or VkDedicatedAllocationMemoryAllocateInfoNV structure with either its image or buffer member set to a value other than VK_NULL_HANDLE.

- **VUID-VkMemoryAllocateInfo-pNext-00640**
  
  If the pNext chain includes a VkExportMemoryAllocateInfo structure, it must not include a VkExportMemoryAllocateInfoNV or VkExportMemoryWin32HandleInfoNV structure.

- **VUID-VkMemoryAllocateInfo-pNext-00641**
  
  If the pNext chain includes a VkImportMemoryWin32HandleInfoKHR structure, it must not include a VkImportMemoryWin32HandleInfoNV structure.

- **VUID-VkMemoryAllocateInfo-allocationSize-01742**
  
  If the parameters define an import operation, the external handle specified was created by the Vulkan API, and the external handle type is VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHR, then the values of allocationSize and memoryTypeIndex must match those specified when the payload being imported was created.

- **VUID-VkMemoryAllocateInfo-None-00643**
  
  If the parameters define an import operation and the external handle specified was created by the Vulkan API, the device mask specified by VkMemoryAllocateFlagsInfo must match that specified when the payload being imported was allocated.

- **VUID-VkMemoryAllocateInfo-None-00644**
  
  If the parameters define an import operation and the external handle specified was created by the Vulkan API, the list of physical devices that comprise the logical device passed to vkAllocateMemory must match the list of physical devices that comprise the logical device on which the payload was originally allocated.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-00645**
  
  If the parameters define an import operation and the external handle is an NT handle or a global share handle created outside of the Vulkan API, the value of memoryTypeIndex must be one of those returned by vkGetMemoryWin32HandlePropertiesKHR.

- **VUID-VkMemoryAllocateInfo-allocationSize-01743**
  
  If the parameters define an import operation, the external handle was created by the Vulkan API, and the external handle type is VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR or VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR, then the values of allocationSize and memoryTypeIndex must match those specified when the payload being imported was created.

- **VUID-VkMemoryAllocateInfo-allocationSize-00647**

Chapter 11. Memory Allocation | 657
If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, allocationSize must match the size specified when creating the Direct3D 12 heap from which the payload was extracted.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-00648**
  If the parameters define an import operation and the external handle is a POSIX file descriptor created outside of the Vulkan API, the value of `memoryTypeIndex` must be one of those returned by `vkGetMemoryFdPropertiesKHR`.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-01872**
  If the protected memory feature is not enabled, the `VkMemoryAllocateInfo::memoryTypeIndex` must not indicate a memory type that reports `VK_MEMORYPROPERTY_PROTECTED_BIT`.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-01744**
  If the parameters define an import operation and the external handle is a host pointer, the value of `memoryTypeIndex` must be one of those returned by `vkGetMemoryHostPointerPropertiesEXT`.

If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORYHANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, allocationSize must be the size returned by `vkGetAndroidHardwareBufferPropertiesANDROID` for the Android hardware buffer.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-02385**
  If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORYHANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `memoryTypeIndex` must be one of those returned by `vkGetAndroidHardwareBufferPropertiesANDROID` for the Android hardware buffer.

- **VUID-VkMemoryAllocateInfo-pNext-01874**
  If the parameters define an import operation and the external handle is a host pointer, `pNext` chain must not include a `VkDedicatedAllocationMemoryAllocateInfoNV` structure with either its `image` or `buffer` field set to a value other than `VK_NULL_HANDLE`.

- **VUID-VkMemoryAllocateInfo-pNext-02806**
  If the parameters define an import operation and the external handle is a host pointer, `pNext` chain must not include a `VkMemoryDedicatedAllocateInfo` structure with either its `image` or `buffer` field set to a value other than `VK_NULL_HANDLE`.

- **VUID-VkMemoryAllocateInfo-allocationSize-02383**
  If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORYHANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, allocationSize must be the size returned by `vkGetAndroidHardwareBufferPropertiesANDROID` for the Android hardware buffer.

- **VUID-VkMemoryAllocateInfo-allocationSize-02384**
  If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORYHANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, and the `pNext` chain does not include a `VkMemoryDedicatedAllocateInfo` structure or `VkMemoryDedicatedAllocateInfo::image` is `VK_NULL_HANDLE`, the Android hardware buffer must have a `AHardwareBuffer_Desc::format` of `AHARDWAREBUFFER_FORMAT_BLOB` and a `AHardwareBuffer_Desc::usage` that includes `AHARDWAREBUFFER_USAGE_GPU_DATA_BUFFER`.

- **VUID-VkMemoryAllocateInfo-memoryTypeIndex-02386**
  If the parameters define an import operation and the external handle type is `VK_EXTERNAL_MEMORYHANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `memoryTypeIndex` must be one of those returned by `vkGetAndroidHardwareBufferPropertiesANDROID` for the Android hardware buffer.
If the parameters do not define an import operation, and the `pNext` chain includes a `VkExportMemoryAllocateInfo` structure with `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` included in its `handleTypes` member, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` structure with `image` not equal to `VK_NULL_HANDLE`, then `allocationSize` must be 0, otherwise `allocationSize` must be greater than 0.

- VUID-VkMemoryAllocateInfo-pNext-02386
  If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, the Android hardware buffer's `AHardwareBuffer::usage` must include at least one of `AHARDWAREBUFFER_USAGE_GPU_FRAMEBUFFER` or `AHARDWAREBUFFER_USAGE_GPU_SAMPLED_IMAGE`.

- VUID-VkMemoryAllocateInfo-pNext-02387
  If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` with `image` that is not `VK_NULL_HANDLE`, the format of `image` must be `VK_FORMAT_UNDEFINED` or the format returned by `vkGetAndroidHardwareBufferPropertiesANDROID` in `VkAndroidHardwareBufferFormatPropertiesANDROID::format` for the Android hardware buffer.

- VUID-VkMemoryAllocateInfo-pNext-02388
  If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` structure with `image` that is not `VK_NULL_HANDLE`, the width, height, and array layer dimensions of `image` and the Android hardware buffer's `AHardwareBuffer_Desc` must be identical.

- VUID-VkMemoryAllocateInfo-pNext-02389
  If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` structure with `image` that is not `VK_NULL_HANDLE`, and the Android hardware buffer's `AHardwareBuffer::usage` includes `AHARDWAREBUFFER_USAGE_GPU_MIPMAP_COMPLETE`, the `image` must have a complete mipmap chain.

- VUID-VkMemoryAllocateInfo-pNext-02586
  If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` structure with `image` that is not `VK_NULL_HANDLE`, and the Android hardware buffer's `AHardwareBuffer::usage` does not include `AHARDWAREBUFFER_USAGE_GPU_MIPMAP_COMPLETE`, the `image` must have exactly one mipmap level.

- VUID-VkMemoryAllocateInfo-pNext-02390
  If the parameters define an import operation, the external handle is an Android hardware buffer, and the `pNext` chain includes a `VkMemoryDedicatedAllocateInfo` structure with `image` that is not `VK_NULL_HANDLE`, each bit set in the usage of `image` must be listed in `AHardwareBuffer Usage Equivalence`, and if there is a corresponding `AHARDWAREBUFFER_USAGE` bit listed that bit must be included in the Android hardware buffer's `AHardwareBuffer_Desc::usage`.

- VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03329
  If `VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress` is not zero,
VkMemoryAllocateFlagsInfo::flags must include VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT

- VUID-VkMemoryAllocateInfo-flags-03330
  If VkMemoryAllocateFlagsInfo::flags includes VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT, the bufferDeviceAddressCaptureReplay feature must be enabled

- VUID-VkMemoryAllocateInfo-flags-03331
  If VkMemoryAllocateFlagsInfo::flags includes VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT, the bufferDeviceAddress feature must be enabled

- VUID-VkMemoryAllocateInfo-pNext-03332
  If the pNext chain includes a VkImportMemoryHostPointerInfoEXT structure, VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress must be zero

- VUID-VkMemoryAllocateInfo-opaqueCaptureAddress-03333
  If the parameters define an import operation, VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress must be zero

- VUID-VkMemoryAllocateInfo-None-04749
  If the parameters define an import operation and the external handle type is VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA, the value of memoryTypeIndex must be an index identifying a memory type from the memoryTypeBits field of the VkMemoryZirconHandlePropertiesFUCHSIA structure populated by a call to vkGetMemoryZirconHandlePropertiesFUCHSIA.

- VUID-VkMemoryAllocateInfo-allocationSize-04750
  If the parameters define an import operation and the external handle type is VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA, the value of allocationSize must be greater than 0 and must be less than or equal to the size of the VMO as determined by zx_vmo_get_size(handle) where handle is the VMO handle to the imported external memory.
Valid Usage (Implicit)

- **VUID-VkMemoryAllocateInfo-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO*

- **VUID-VkMemoryAllocateInfo-pNext-pNext**
  
  Each *pNext* member of any structure (including this one) in the *pNext* chain *must* be either NULL or a pointer to a valid instance of VkDedicatedAllocationMemoryAllocateInfoNV, VkExportMemoryAllocateInfo, VkExportMemoryWin32HandleInfoKHR, VkExportMemoryWin32HandleInfoNV, VkExportMemoryWin32HandleInfoNV, VkExportMemoryWin32HandleInfoKHR, VkImportAndroidHardwareBufferInfoANDROID, VkImportMemoryFdInfoKHR, VkImportMemoryHostPointerInfoEXT, VkImportMemoryZirconHandleInfoFUCHSIA, VkMemoryAllocateFlagsInfo, VkMemoryDedicatedAllocateInfo, VkMemoryOpaqueCaptureAddressAllocateInfo, or VkMemoryPriorityAllocateInfoEXT

- **VUID-VkMemoryAllocateInfo-sType-unique**
  
  The *sType* value of each struct in the *pNext* chain *must* be unique

If the *pNext* chain includes a VkMemoryDedicatedAllocateInfo structure, then that structure includes a handle of the sole buffer or image resource that the memory can be bound to.

The VkMemoryDedicatedAllocateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkMemoryDedicatedAllocateInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkImage            image;
    VkBuffer           buffer;
} VkMemoryDedicatedAllocateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_dedicated_allocation
typedef VkMemoryDedicatedAllocateInfo VkMemoryDedicatedAllocateInfoKHR;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *image* is VK_NULL_HANDLE or a handle of an image which this memory will be bound to.
- *buffer* is VK_NULL_HANDLE or a handle of a buffer which this memory will be bound to.
Valid Usage

- **VUID-VkMemoryDedicatedAllocateInfo-image-01432**
  At least one of `image` and `buffer` must be `VK_NULL_HANDLE`

- **VUID-VkMemoryDedicatedAllocateInfo-image-02964**
  If `image` is not `VK_NULL_HANDLE` and the memory is not an imported Android Hardware Buffer, `VkMemoryAllocateInfo::allocationSize` must equal the `VkMemoryRequirements::size` of the image

- **VUID-VkMemoryDedicatedAllocateInfo-image-01434**
  If `image` is not `VK_NULL_HANDLE`, `image` must have been created without `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` set in `VkImageCreateInfo::flags`

- **VUID-VkMemoryDedicatedAllocateInfo-buffer-02965**
  If `buffer` is not `VK_NULL_HANDLE` and the memory is not an imported Android Hardware Buffer, `VkMemoryAllocateInfo::allocationSize` must equal the `VkMemoryRequirements::size` of the buffer

- **VUID-VkMemoryDedicatedAllocateInfo-buffer-01436**
  If `buffer` is not `VK_NULL_HANDLE`, `buffer` must have been created without `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` set in `VkBufferCreateInfo::flags`

- **VUID-VkMemoryDedicatedAllocateInfo-image-01876**
  If `image` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, and the external handle was created by the Vulkan API, then the memory being imported must also be a dedicated image allocation and `image` must be identical to the image associated with the imported memory

- **VUID-VkMemoryDedicatedAllocateInfo-buffer-01877**
  If `buffer` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, and the external handle was created by the Vulkan API, then the memory being imported must also be a dedicated buffer allocation and `buffer` must be identical to the buffer associated with the imported memory

- **VUID-VkMemoryDedicatedAllocateInfo-image-01878**
  If `image` is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation with handle type `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT`, the memory being imported must also be a dedicated image allocation and `image` must be identical to the image associated with the imported memory
If buffer is not VK_NULL_HANDLE and VkMemoryAllocateInfo defines a memory import operation with handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT, the memory being imported must also be a dedicated buffer allocation and buffer must be identical to the buffer associated with the imported memory.

If image is not VK_NULL_HANDLE, image must not have been created with VK_IMAGE_CREATE_DISJOINT_BIT set in VkImageCreateInfo::flags.

If image is not VK_NULL_HANDLE and VkMemoryAllocateInfo defines a memory import operation with handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA, the memory being imported must also be a dedicated image allocation and image must be identical to the image associated with the imported memory.

If buffer is not VK_NULL_HANDLE and VkMemoryAllocateInfo defines a memory import operation with handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA, the memory being imported must also be a dedicated buffer allocation and buffer must be identical to the buffer associated with the imported memory.

Valid Usage (Implicit)

- **sType** must be VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO
- If image is not VK_NULL_HANDLE, image must be a valid VkImage handle
- If buffer is not VK_NULL_HANDLE, buffer must be a valid VkBuffer handle
- Both of buffer, and image that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

If the pNext chain includes a VkDedicatedAllocationMemoryAllocateInfoNV structure, then that structure includes a handle of the sole buffer or image resource that the memory can be bound to.

The VkDedicatedAllocationMemoryAllocateInfoNV structure is defined as:

```c
// Provided by VK_NV_dedicated_allocation
typedef struct VkDedicatedAllocationMemoryAllocateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
    VkBuffer buffer;
} VkDedicatedAllocationMemoryAllocateInfoNV;
```
• **sType** is the type of this structure.
• **pNext** is `NULL` or a pointer to a structure extending this structure.
• **image** is `VK_NULL_HANDLE` or a handle of an image which this memory will be bound to.
• **buffer** is `VK_NULL_HANDLE` or a handle of a buffer which this memory will be bound to.

### Valid Usage

- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-image-00649](#)
  - At least one of **image** and **buffer** must be `VK_NULL_HANDLE`
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-image-00650](#)
  - If **image** is not `VK_NULL_HANDLE`, the image must have been created with `VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-buffer-00651](#)
  - If **buffer** is not `VK_NULL_HANDLE`, the buffer must have been created with `VkDedicatedAllocationBufferCreateInfoNV::dedicatedAllocation` equal to `VK_TRUE`
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-image-00652](#)
  - If **image** is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo::allocationSize` must equal the `VkMemoryRequirements::size` of the image
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-buffer-00653](#)
  - If **buffer** is not `VK_NULL_HANDLE`, `VkMemoryAllocateInfo::allocationSize` must equal the `VkMemoryRequirements::size` of the buffer
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-image-00654](#)
  - If **image** is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation, the memory being imported must also be a dedicated image allocation and **image** must be identical to the image associated with the imported memory
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-buffer-00655](#)
  - If **buffer** is not `VK_NULL_HANDLE` and `VkMemoryAllocateInfo` defines a memory import operation, the memory being imported must also be a dedicated buffer allocation and **buffer** must be identical to the buffer associated with the imported memory

### Valid Usage (Implicit)

- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-sType-sType](#)
  - **sType** must be `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV`
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-image-parameter](#)
  - If **image** is not `VK_NULL_HANDLE`, **image** must be a valid `VkImage` handle
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-buffer-parameter](#)
  - If **buffer** is not `VK_NULL_HANDLE`, **buffer** must be a valid `VkBuffer` handle
- [VUID-VkDedicatedAllocationMemoryAllocateInfoNV-commonparent](#)
  - Both of **buffer**, and **image** that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`
If the `pNext` chain includes a `VkMemoryPriorityAllocateInfoEXT` structure, then that structure includes a priority for the memory.

The `VkMemoryPriorityAllocateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_memory_priority
typedef struct VkMemoryPriorityAllocateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    float priority;
} VkMemoryPriorityAllocateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `priority` is a floating-point value between 0 and 1, indicating the priority of the allocation relative to other memory allocations. Larger values are higher priority. The granularity of the priorities is implementation-dependent.

Memory allocations with higher priority may be more likely to stay in device-local memory when the system is under memory pressure.

If this structure is not included, it is as if the `priority` value were 0.5.

**Valid Usage**

- VUID-VkMemoryPriorityAllocateInfoEXT-priority-02602
  
  `priority` must be between 0 and 1, inclusive

**Valid Usage (Implicit)**

- VUID-VkMemoryPriorityAllocateInfoEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_MEMORY_PRIORITY_ALLOCATE_INFO_EXT`

When allocating memory whose payload may be exported to another process or Vulkan instance, add a `VkExportMemoryAllocateInfo` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure, specifying the handle types that may be exported.

The `VkExportMemoryAllocateInfo` structure is defined as:
or the equivalent

```c
// Provided by VK_KHR_external_memory
typedef VkExportMemoryAllocateInfo VkExportMemoryAllocateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more memory handle types the application can export from the resulting allocation. The application can request multiple handle types for the same allocation.

### Valid Usage

- VUID-VkExportMemoryAllocateInfo-handleTypes-00656
  The bits in `handleTypes` must be supported and compatible, as reported by `VkExternalImageFormatProperties` or `VkExternalBufferProperties`

### Valid Usage (Implicit)

- VUID-VkExportMemoryAllocateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO`

- VUID-VkExportMemoryAllocateInfo-handleTypes-parameter
  `handleTypes` must be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values

To specify additional attributes of NT handles exported from a memory object, add a `VkExportMemoryWin32HandleInfoKHR` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure. The `VkExportMemoryWin32HandleInfoKHR` structure is defined as:
typedef struct VkExportMemoryWin32HandleInfoKHR {
    VkStructureType       sType;
    const void*           pNext;
    const SECURITY_ATTRIBUTES* pAttributes;
    DWORD                  dwAccess;
    LPCWSTR                name;
} VkExportMemoryWin32HandleInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pAttributes** is a pointer to a Windows SECURITY_ATTRIBUTES structure specifying security attributes of the handle.
- **dwAccess** is a DWORD specifying access rights of the handle.
- **name** is a null-terminated UTF-16 string to associate with the payload referenced by NT handles exported from the created memory.

If **VkExportMemoryAllocateInfo** is not present in the same **pNext** chain, this structure is ignored.

If **VkExportMemoryAllocateInfo** is present in the **pNext** chain of **VkMemoryAllocateInfo** with a Windows **handleType**, but either **VkExportMemoryWin32HandleInfoKHR** is not present in the **pNext** chain, or if it is but **pAttributes** is set to NULL, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”\(^1\). Further, if the structure is not present, the access rights used depend on the handle type.

For handles of the following types:

```
VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT
```

The implementation **must** ensure the access rights allow read and write access to the memory.

For handles of the following types:

```
VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT
```

The access rights **must be:**

```
GENERIC_ALL
```

Valid Usage

• VUID-VkExportMemoryWin32HandleInfoKHR-handleTypes-00657

If `VkExportMemoryAllocateInfo::handleTypes` does not include `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT, VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT, VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT, or VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, a `VkExportMemoryWin32HandleInfoKHR` structure must not be included in the `pNext` chain of `VkMemoryAllocateInfo`.

Valid Usage (Implicit)

• VUID-VkExportMemoryWin32HandleInfoKHR-sType-sType

  `sType` must be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR`

• VUID-VkExportMemoryWin32HandleInfoKHR-pAttributes-parameter

  If `pAttributes` is not `NULL`, `pAttributes` must be a valid pointer to a valid `SECURITY_ATTRIBUTES` value.

To import memory from a Windows handle, add a `VkImportMemoryWin32HandleInfoKHR` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure.

The `VkImportMemoryWin32HandleInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_memory_win32
typedef struct VkImportMemoryWin32HandleInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    HANDLE handle;
    LPCWSTR name;
} VkImportMemoryWin32HandleInfoKHR;
```

• `sType` is the type of this structure.

• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `handleType` specifies the type of `handle` or `name`.

• `handle` is the external handle to import, or `NULL`.

• `name` is a null-terminated UTF-16 string naming the payload to import, or `NULL`.

Importing memory object payloads from Windows handles does not transfer ownership of the handle to the Vulkan implementation. For handle types defined as NT handles, the application must release handle ownership using the `CloseHandle` system call when the handle is no longer needed. For handle types defined as NT handles, the imported memory object holds a reference to its payload.
**Non-NT handle import operations do not add a reference to their associated payload. If the original object owning the payload is destroyed, all resources and handles sharing that payload will become invalid.**

Applications **can** import the same payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation **must** create a distinct `VkDeviceMemory` object.

### Valid Usage

- **VUID-VkImportMemoryWin32HandleInfoKHR-handleType-00658**
  - *If `handleType` is not 0, it must be supported for import, as reported by `VkExternalImageFormatProperties` or `VkExternalBufferProperties`*

- **VUID-VkImportMemoryWin32HandleInfoKHR-handle-00659**
  - *The memory from which `handle` was exported, or the memory named by `name` must have been created on the same underlying physical device as `device`*

- **VUID-VkImportMemoryWin32HandleInfoKHR-handleType-00660**
  - *If `handleType` is not 0, it must be defined as an NT handle or a global share handle*

- **VUID-VkImportMemoryWin32HandleInfoKHR-handleType-01439**
  - *If `handleType` is not `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT`, `name` must be NULL*

- **VUID-VkImportMemoryWin32HandleInfoKHR-handleType-01440**
  - *If `handleType` is not 0 and `handle` is NULL, `name` must name a valid memory resource of the type specified by `handleType`*

- **VUID-VkImportMemoryWin32HandleInfoKHR-handle-01441**
  - *If `handle` is not NULL, `name` must be NULL*

- **VUID-VkImportMemoryWin32HandleInfoKHR-handleType-01518**
  - *If `handle` is not NULL, it must obey any requirements listed for `handleType` in external memory handle types compatibility*

- **VUID-VkImportMemoryWin32HandleInfoKHR-name-01519**
  - *If `name` is not NULL, it must obey any requirements listed for `handleType` in external memory handle types compatibility*
Valid Usage (Implicit)

- VUID-VkImportMemoryWin32HandleInfoKHR-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHR

- VUID-VkImportMemoryWin32HandleInfoKHR-handleType-parameter
  
  If handleType is not 0, handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

To export a Windows handle representing the payload of a Vulkan device memory object, call:

```c
// Provided by VK_KHR_external_memory_win32
VkResult vkGetMemoryWin32HandleKHR(
    VkDevice device, 
    const VkMemoryGetWin32HandleInfoKHR* pGetWin32HandleInfo, 
    HANDLE* pHandle);
```

- device is the logical device that created the device memory being exported.
- pGetWin32HandleInfo is a pointer to a VkMemoryGetWin32HandleInfoKHR structure containing parameters of the export operation.
- pHandle will return the Windows handle representing the payload of the device memory object.

For handle types defined as NT handles, the handles returned by vkGetMemoryWin32HandleKHR are owned by the application and hold a reference to their payload. To avoid leaking resources, the application must release ownership of them using the CloseHandle system call when they are no longer needed.

**Note**

Non-NT handle types do not add a reference to their associated payload. If the original object owning the payload is destroyed, all resources and handles sharing that payload will become invalid.

Valid Usage (Implicit)

- VUID-vkGetMemoryWin32HandleKHR-device-parameter
  
  device must be a valid VkDevice handle

- VUID-vkGetMemoryWin32HandleKHR-pGetWin32HandleInfo-parameter
  
  pGetWin32HandleInfo must be a valid pointer to a valid VkMemoryGetWin32HandleInfoKHR structure

- VUID-vkGetMemoryWin32HandleKHR-pHandle-parameter
  
  pHandle must be a valid pointer to a HANDLE value
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The `VkMemoryGetWin32HandleInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_memory_win32
typedef struct VkMemoryGetWin32HandleInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkMemoryGetWin32HandleInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memory` is the memory object from which the handle will be exported.
- `handleType` is the type of handle requested.

The properties of the handle returned depend on the value of `handleType`. See `VkExternalMemoryHandleTypeFlagBits` for a description of the properties of the defined external memory handle types.

Valid Usage

- VUID-VkMemoryGetWin32HandleInfoKHR-handleType-00662
  `handleType` must have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created

- VUID-VkMemoryGetWin32HandleInfoKHR-handleType-00663
  If `handleType` is defined as an NT handle, `vkGetMemoryWin32HandleKHR` must be called no more than once for each valid unique combination of `memory` and `handleType`

- VUID-VkMemoryGetWin32HandleInfoKHR-handleType-00664
  `handleType` must be defined as an NT handle or a global share handle
Valid Usage (Implicit)

- **VUID-VkMemoryGetWin32HandleInfoKHR-sType-sType**
  
  sType must be VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR

- **VUID-VkMemoryGetWin32HandleInfoKHR-pNext-pNext**
  
  pNext must be NULL

- **VUID-VkMemoryGetWin32HandleInfoKHR-memory-parameter**
  
  memory must be a valid VkDeviceMemory handle

- **VUID-VkMemoryGetWin32HandleInfoKHR-handleType-parameter**
  
  handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

Windows memory handles compatible with Vulkan may also be created by non-Vulkan APIs using methods beyond the scope of this specification. To determine the correct parameters to use when importing such handles, call:

```c
// Provided by VK_KHR_external_memory_win32
VkResult vkGetMemoryWin32HandlePropertiesKHR(
    VkDevice                                    device,
    VkExternalMemoryHandleTypeFlagBits          handleType,
    HANDLE                                      handle,
    VkMemoryWin32HandlePropertiesKHR*           pMemoryWin32HandleProperties);
```

- **device** is the logical device that will be importing handle.
- **handleType** is the type of the handle handle.
- **handle** is the handle which will be imported.
- **pMemoryWin32HandleProperties** will return properties of handle.

Valid Usage

- **VUID-vkGetMemoryWin32HandlePropertiesKHR-handle-00665**
  
  handle must be an external memory handle created outside of the Vulkan API

- **VUID-vkGetMemoryWin32HandlePropertiesKHR-handleType-00666**
  
  handleType must not be one of the handle types defined as opaque
Valid Usage (Implicit)

- VUID-vkGetMemoryWin32HandlePropertiesKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetMemoryWin32HandlePropertiesKHR-handleType-parameter
  handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

- VUID-vkGetMemoryWin32HandlePropertiesKHR-pMemoryWin32HandleProperties-parameter
  pMemoryWin32HandleProperties must be a valid pointer to a
  VkMemoryWin32HandlePropertiesKHR structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkMemoryWin32HandlePropertiesKHR structure returned is defined as:

```c
// Provided by VK_KHR_external_memory_win32
typedef struct VkMemoryWin32HandlePropertiesKHR {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           memoryTypeBits;
} VkMemoryWin32HandlePropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **memoryTypeBits** is a bitmask containing one bit set for every memory type which the specified windows handle can be imported as.

Valid Usage (Implicit)

- VUID-VkMemoryWin32HandlePropertiesKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHR

- VUID-VkMemoryWin32HandlePropertiesKHR-pNext-pNext
  pNext must be NULL

To import memory from a POSIX file descriptor handle, add a VkImportMemoryFdInfoKHR structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkImportMemoryFdInfoKHR
structure is defined as:

```c
// Provided by VK_KHR_external_memory_fd
typedef struct VkImportMemoryFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    int fd;
} VkImportMemoryFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleType` specifies the handle type of `fd`.
- `fd` is the external handle to import.

Importing memory from a file descriptor transfers ownership of the file descriptor from the application to the Vulkan implementation. The application must not perform any operations on the file descriptor after a successful import. The imported memory object holds a reference to its payload.

Applications can import the same payload into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation must create a distinct `VkDeviceMemory` object.

### Valid Usage

- **VUID-VkImportMemoryFdInfoKHR-handleType-00667**  
  If `handleType` is not `0`, it must be supported for import, as reported by  
  `VkExternalImageFormatProperties` or `VkExternalBufferProperties`.

- **VUID-VkImportMemoryFdInfoKHR-fd-00668**  
  The memory from which `fd` was exported must have been created on the same underlying physical device as `device`.

- **VUID-VkImportMemoryFdInfoKHR-handleType-00669**  
  If `handleType` is not `0`, it must be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT`.

- **VUID-VkImportMemoryFdInfoKHR-handleType-00670**  
  If `handleType` is not `0`, `fd` must be a valid POSIX file descriptor handle.

- **VUID-VkImportMemoryFdInfoKHR-fd-01746**  
  The memory represented by `fd` must have been created from a physical device and driver that is compatible with `device` and `handleType`, as described in External memory handle types compatibility.

- **VUID-VkImportMemoryFdInfoKHR-fd-01520**  
  `fd` must obey any requirements listed for `handleType` in external memory handle types compatibility.
To export a POSIX file descriptor referencing the payload of a Vulkan device memory object, call:

```c
// Provided by VK_KHR_external_memory_fd
VkResult vkGetMemoryFdKHR(
    VkDevice                                    device,
    const VkMemoryGetFdInfoKHR*                 pGetFdInfo,
    int*                                        pFd);
```

- `device` is the logical device that created the device memory being exported.
- `pGetFdInfo` is a pointer to a `VkMemoryGetFdInfoKHR` structure containing parameters of the export operation.
- `pFd` will return a file descriptor referencing the payload of the device memory object.

Each call to `vkGetMemoryFdKHR` must create a new file descriptor holding a reference to the memory object’s payload and transfer ownership of the file descriptor to the application. To avoid leaking resources, the application must release ownership of the file descriptor using the `close` system call when it is no longer needed, or by importing a Vulkan memory object from it. Where supported by the operating system, the implementation must set the file descriptor to be closed automatically when an `execve` system call is made.

---

**Valid Usage (Implicit)**

- VUID-vkGetMemoryFdKHR-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkGetMemoryFdKHR-pGetFdInfo-parameter
  - `pGetFdInfo` must be a valid pointer to a valid `VkMemoryGetFdInfoKHR` structure
- VUID-vkGetMemoryFdKHR-pFd-parameter
  - `pFd` must be a valid pointer to an `int` value
Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The `VkMemoryGetFdInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_external_memory_fd
typedef struct VkMemoryGetFdInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkMemoryGetFdInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memory` is the memory object from which the handle will be exported.
- `handleType` is the type of handle requested.

The properties of the file descriptor exported depend on the value of `handleType`. See `VkExternalMemoryHandleTypeFlagBits` for a description of the properties of the defined external memory handle types.

**Note**
The size of the exported file **may** be larger than the size requested by `VkMemoryAllocateInfo::allocationSize`. If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`, then the application **can** query the file’s actual size with `lseek(2)`.

Valid Usage

- VUID-VkMemoryGetFdInfoKHR-handleType-00671
  - `handleType` must have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created

- VUID-VkMemoryGetFdInfoKHR-handleType-00672
  - `handleType` must be defined as a POSIX file descriptor handle
Valid Usage (Implicit)

- VUID-VkMemoryGetFdInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR

- VUID-VkMemoryGetFdInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkMemoryGetFdInfoKHR-memory-parameter
  memory must be a valid VkDeviceMemory handle

- VUID-VkMemoryGetFdInfoKHR-handleType-parameter
  handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

POSIX file descriptor memory handles compatible with Vulkan may also be created by non-Vulkan APIs using methods beyond the scope of this specification. To determine the correct parameters to use when importing such handles, call:

```c
// Provided by VK_KHR_external_memory_fd
VkResult vkGetMemoryFdPropertiesKHR(
    VkDevice                                    device,         
    VkExternalMemoryHandleTypeFlagBits          handleType,    
    int                                        fd,           
    VkMemoryFdPropertiesKHR*                    pMemoryFdProperties);
```

- `device` is the logical device that will be importing `fd`.
- `handleType` is the type of the handle `fd`.
- `fd` is the handle which will be imported.
- `pMemoryFdProperties` is a pointer to a VkMemoryFdPropertiesKHR structure in which the properties of the handle `fd` are returned.

Valid Usage

- VUID-vkGetMemoryFdPropertiesKHR-fd-00673
  `fd` must be an external memory handle created outside of the Vulkan API

- VUID-vkGetMemoryFdPropertiesKHR-handleType-00674
  `handleType` must not be VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHR
Valid Usage (Implicit)

- VUID-vkGetMemoryFdPropertiesKHR-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetMemoryFdPropertiesKHR-handleType-parameter
dhandleType must be a valid VkExternalMemoryHandleTypeFlagBits value

- VUID-vkGetMemoryFdPropertiesKHR-pMemoryFdProperties-parameter
pMemoryFdProperties must be a valid pointer to a VkMemoryFdPropertiesKHR structure

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkMemoryFdPropertiesKHR structure returned is defined as:

```cpp
// Provided by VK_KHR_external_memory_fd
typedef struct VkMemoryFdPropertiesKHR {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           memoryTypeBits;
} VkMemoryFdPropertiesKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- memoryTypeBits is a bitmask containing one bit set for every memory type which the specified file descriptor can be imported as.

Valid Usage (Implicit)

- VUID-VkMemoryFdPropertiesKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR

- VUID-VkMemoryFdPropertiesKHR-pNext-pNext
  pNext must be NULL

To import memory from a host pointer, add a VkImportMemoryHostPointerInfoEXT structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkImportMemoryHostPointerInfoEXT structure is defined as:
// Provided by VK_EXT_external_memory_host

typedef struct VkImportMemoryHostPointerInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlagBits handleType;
    void* pHostPointer;
} VkImportMemoryHostPointerInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleType** specifies the handle type.
- **pHostPointer** is the host pointer to import from.

Importing memory from a host pointer shares ownership of the memory between the host and the Vulkan implementation. The application can continue to access the memory through the host pointer but it is the application’s responsibility to synchronize device and non-device access to the payload as defined in Host Access to Device Memory Objects.

Applications can import the same payload into multiple instances of Vulkan and multiple times into a given Vulkan instance. However, implementations may fail to import the same payload multiple times into a given physical device due to platform constraints.

Importing memory from a particular host pointer may not be possible due to additional platform-specific restrictions beyond the scope of this specification in which case the implementation must fail the memory import operation with the error code VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR.

Whether device memory objects imported from a host pointer hold a reference to their payload is undefined. As such, the application must ensure that the imported memory range remains valid and accessible for the lifetime of the imported memory object.
Valid Usage

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01747**
  If `handleType` is not 0, it **must** be supported for import, as reported in `VkExternalMemoryProperties`

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01748**
  If `handleType` is not 0, it **must** be `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`

- **VUID-VkImportMemoryHostPointerInfoEXT-pHostPointer-01749**
  `pHostPointer` **must** be a pointer aligned to an integer multiple of `VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment`

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01750**
  If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT`, `pHostPointer` **must** be a pointer to `allocationSize` number of bytes of host memory, where `allocationSize` is the member of the `VkMemoryAllocateInfo` structure this structure is chained to

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-01751**
  If `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT`, `pHostPointer` **must** be a pointer to `allocationSize` number of bytes of host mapped foreign memory, where `allocationSize` is the member of the `VkMemoryAllocateInfo` structure this structure is chained to

Valid Usage (Implicit)

- **VUID-VkImportMemoryHostPointerInfoEXT-sType-sType**
  `sType` **must** be `VK_STRUCTURE_TYPE_IMPORT_MEMORY_HOST_POINTER_INFO_EXT`

- **VUID-VkImportMemoryHostPointerInfoEXT-handleType-parameter**
  `handleType` **must** be a valid `VkExternalMemoryHandleTypeFlagBits` value

To determine the correct parameters to use when importing host pointers, call:

```c
VkResult vkGetMemoryHostPointerPropertiesEXT(
    VkDevice                                    device,
    VkExternalMemoryHandleTypeFlagBits          handleType,
    const void*                                 pHostPointer,
    VkMemoryHostPointerPropertiesEXT*           pMemoryHostPointerProperties);
```

- `device` is the logical device that will be importing `pHostPointer`
- `handleType` is the type of the handle `pHostPointer`
- `pHostPointer` is the host pointer to import from
- `pMemoryHostPointerProperties` is a pointer to a `VkMemoryHostPointerPropertiesEXT` structure in
which the host pointer properties are returned.

**Valid Usage**

- **VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-01752**
  
  *handleType must be* VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT or VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT

- **VUID-vkGetMemoryHostPointerPropertiesEXT-pHostPointer-01753**
  
  *pHostPointer must be a pointer aligned to an integer multiple of VkPhysicalDeviceExternalMemoryHostPropertiesEXT::minImportedHostPointerAlignment*

- **VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-01754**
  
  *If handleType is VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT, pHostPointer must be a pointer to host memory*

- **VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-01755**
  
  *If handleType is VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT, pHostPointer must be a pointer to host mapped foreign memory*

**Valid Usage (Implicit)**

- **VUID-vkGetMemoryHostPointerPropertiesEXT-device-parameter**
  
  *device must be a valid VkDevice handle*

- **VUID-vkGetMemoryHostPointerPropertiesEXT-handleType-parameter**
  
  *handleType must be a valid VkExternalMemoryHandleTypeFlagBits value*

- **VUID-vkGetMemoryHostPointerPropertiesEXT-pMemoryHostPointerProperties-parameter**
  
  *pMemoryHostPointerProperties must be a valid pointer to a VkMemoryHostPointerPropertiesEXT structure*

**Return Codes**

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkMemoryHostPointerPropertiesEXT structure is defined as:
typedef struct VkMemoryHostPointerPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t memoryTypeBits;
} VkMemoryHostPointerPropertiesEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **memoryTypeBits** is a bitmask containing one bit set for every memory type which the specified host pointer can be imported as.

The value returned by `memoryTypeBits` must only include bits that identify memory types which are host visible.

**Valid Usage (Implicit)**

- VUID-VkMemoryHostPointerPropertiesEXT-sType-sType
  - **sType** must be `VK_STRUCTURE_TYPE_MEMORY_HOST_POINTER_PROPERTIES_EXT`
- VUID-VkMemoryHostPointerPropertiesEXT-pNext-pNext
  - **pNext** must be NULL

To import memory created outside of the current Vulkan instance from an Android hardware buffer, add a `VkImportAndroidHardwareBufferInfoANDROID` structure to the pNext chain of the `VkMemoryAllocateInfo` structure. The `VkImportAndroidHardwareBufferInfoANDROID` structure is defined as:

typedef struct VkImportAndroidHardwareBufferInfoANDROID {
    VkStructureType sType;
    const void* pNext;
    struct AHardwareBuffer* buffer;
} VkImportAndroidHardwareBufferInfoANDROID;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **buffer** is the Android hardware buffer to import.

If the `vkAllocateMemory` command succeeds, the implementation must acquire a reference to the imported hardware buffer, which it must release when the device memory object is freed. If the command fails, the implementation must not retain a reference.
Valid Usage

• VUID-VkImportAndroidHardwareBufferInfoANDROID-buffer-01880
  If buffer is not NULL, Android hardware buffers must be supported for import, as reported by VkExternalImageFormatProperties or VkExternalBufferProperties.

• VUID-VkImportAndroidHardwareBufferInfoANDROID-buffer-01881
  If buffer is not NULL, it must be a valid Android hardware buffer object with AHardwareBuffer_Desc::usage compatible with Vulkan as described in Android Hardware Buffers.

Valid Usage (Implicit)

• VUID-VkImportAndroidHardwareBufferInfoANDROID-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMPORT_ANDROID_HARDWARE_BUFFER_INFO_ANDROID.

• VUID-VkImportAndroidHardwareBufferInfoANDROID-buffer-parameter
  buffer must be a valid pointer to an AHardwareBuffer value.

To export an Android hardware buffer referencing the payload of a Vulkan device memory object, call:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VkResult vkGetMemoryAndroidHardwareBufferANDROID(
    VkDevice device,
    const VkMemoryGetAndroidHardwareBufferInfoANDROID* pInfo,
    struct AHardwareBuffer** pBuffer);
```

• device is the logical device that created the device memory being exported.

• pInfo is a pointer to a VkMemoryGetAndroidHardwareBufferInfoANDROID structure containing parameters of the export operation.

• pBuffer will return an Android hardware buffer referencing the payload of the device memory object.

Each call to vkGetMemoryAndroidHardwareBufferANDROID must return an Android hardware buffer with a new reference acquired in addition to the reference held by the VkDeviceMemory. To avoid leaking resources, the application must release the reference by calling AHardwareBuffer_release when it is no longer needed. When called with the same handle in VkMemoryGetAndroidHardwareBufferInfoANDROID::memory, vkGetMemoryAndroidHardwareBufferANDROID must return the same Android hardware buffer object. If the device memory was created by importing an Android hardware buffer, vkGetMemoryAndroidHardwareBufferANDROID must return that same Android hardware buffer object.
Valid Usage (Implicit)

- **VUID-vkGetMemoryAndroidHardwareBufferANDROID-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkGetMemoryAndroidHardwareBufferANDROID-pInfo-parameter**
  
  `pInfo` must be a valid pointer to a valid `VkMemoryGetAndroidHardwareBufferInfoANDROID` structure

- **VUID-vkGetMemoryAndroidHardwareBufferANDROID-pBuffer-parameter**
  
  `pBuffer` must be a valid pointer to a valid pointer to an `AHardwareBuffer` value

Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkMemoryGetAndroidHardwareBufferInfoANDROID` structure is defined as:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
typedef struct VkMemoryGetAndroidHardwareBufferInfoANDROID {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
} VkMemoryGetAndroidHardwareBufferInfoANDROID;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memory` is the memory object from which the Android hardware buffer will be exported.

Valid Usage

- **VUID-VkMemoryGetAndroidHardwareBufferInfoANDROID-handleTypes-01882**
  
  `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` must have been included in `VkExportMemoryAllocateInfo::handleTypes` when `memory` was created

- **VUID-VkMemoryGetAndroidHardwareBufferInfoANDROID-pNext-01883**
  
  If the `pNext` chain of the `VkMemoryAllocateInfo` used to allocate `memory` included a `VkMemoryDedicatedAllocateInfo` with non-`NULL` image member, then that `image` must already be bound to `memory`
Valid Usage (Implicit)

- VUID-VkMemoryGetAndroidHardwareBufferInfoANDROID-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_GET_ANDROID_HARDWARE_BUFFER_INFO_ANDROID

- VUID-VkMemoryGetAndroidHardwareBufferInfoANDROID-pNext-pNext
  pNext must be NULL

- VUID-VkMemoryGetAndroidHardwareBufferInfoANDROID-memory-parameter
  memory must be a valid VkDeviceMemory handle

To determine the memory parameters to use when importing an Android hardware buffer, call:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
VkResult vkGetAndroidHardwareBufferPropertiesANDROID(
    VkDevice                                    device,
    const struct AHardwareBuffer*               buffer,
    VkAndroidHardwareBufferPropertiesANDROID*   pProperties);
```

- device is the logical device that will be importing buffer.
- buffer is the Android hardware buffer which will be imported.
- pProperties is a pointer to a VkAndroidHardwareBufferPropertiesANDROID structure in which the properties of buffer are returned.

Valid Usage

- VUID-vkGetAndroidHardwareBufferPropertiesANDROID-buffer-01884
  buffer must be a valid Android hardware buffer object with at least one of the
  AHWAREBUFFER_USAGE_GPU_* flags in its AHardwareBuffer_Desc::usage

Valid Usage (Implicit)

- VUID-vkGetAndroidHardwareBufferPropertiesANDROID-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetAndroidHardwareBufferPropertiesANDROID-buffer-parameter
  buffer must be a valid pointer to a valid AHardwareBuffer value

- VUID-vkGetAndroidHardwareBufferPropertiesANDROID-pProperties-parameter
  pProperties must be a valid pointer to a VkAndroidHardwareBufferPropertiesANDROID structure
Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR

The `VkAndroidHardwareBufferPropertiesANDROID` structure returned is defined as:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
typedef structVkAndroidHardwareBufferPropertiesANDROID {
    VkStructureType    sType;
    void*              pNext;
    VkDeviceSize       allocationSize;
    uint32_t           memoryTypeBits;
} VkAndroidHardwareBufferPropertiesANDROID;
```

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `allocationSize` is the size of the external memory
• `memoryTypeBits` is a bitmask containing one bit set for every memory type which the specified Android hardware buffer can be imported as.

Valid Usage (Implicit)

• `VUID-VkAndroidHardwareBufferPropertiesANDROID-sType-sType`
  `sType` must be `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_PROPERTIESANDROID`

• `VUID-VkAndroidHardwareBufferPropertiesANDROID-pNext-pNext`
  `pNext` must be `NULL` or a pointer to a valid instance of `VkAndroidHardwareBufferFormatPropertiesANDROID`

• `VUID-VkAndroidHardwareBufferPropertiesANDROID-sType-unique`
  The `sType` value of each struct in the `pNext` chain must be unique

To obtain format properties of an Android hardware buffer, include a `VkAndroidHardwareBufferFormatPropertiesANDROID` structure in the `pNext` chain of the `VkAndroidHardwareBufferPropertiesANDROID` structure passed to `vkGetAndroidHardwareBufferPropertiesANDROID`. This structure is defined as:
typedef struct VkAndroidHardwareBufferFormatPropertiesANDROID {
    VkStructureType sType;
    void* pNext;
    VkFormat format;
    uint64_t externalFormat;
    VkFormatFeatureFlags formatFeatures;
    VkComponentMapping samplerYcbcrConversionComponents;
    VkSamplerYcbcrModelConversion suggestedYcbcrModel;
    VkSamplerYcbcrRange suggestedYcbcrRange;
    VkChromaLocation suggestedXChromaOffset;
    VkChromaLocation suggestedYChromaOffset;
} VkAndroidHardwareBufferFormatPropertiesANDROID;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **format** is the Vulkan format corresponding to the Android hardware buffer’s format, or **VK_FORMAT_UNDEFINED** if there is not an equivalent Vulkan format.

• **externalFormat** is an implementation-defined external format identifier for use with **VkExternalFormatANDROID**. It **must** not be zero.

• **formatFeatures** describes the capabilities of this external format when used with an image bound to memory imported from **buffer**.

• **samplerYcbcrConversionComponents** is the component swizzle that **should** be used in **VkSamplerYcbcrConversionCreateInfo**.

• **suggestedYcbcrModel** is a suggested color model to use in the **VkSamplerYcbcrConversionCreateInfo**.

• **suggestedYcbcrRange** is a suggested numerical value range to use in **VkSamplerYcbcrConversionCreateInfo**.

• **suggestedXChromaOffset** is a suggested X chroma offset to use in **VkSamplerYcbcrConversionCreateInfo**.

• **suggestedYChromaOffset** is a suggested Y chroma offset to use in **VkSamplerYcbcrConversionCreateInfo**.

If the Android hardware buffer has one of the formats listed in the **Format Equivalence table**, then **format** must have the equivalent Vulkan format listed in the table. Otherwise, **format** may be **VK_FORMAT_UNDEFINED**, indicating the Android hardware buffer can only be used with an external format.

The **formatFeatures** member **must** include **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** and at least one of **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT** or **VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT**, and **should** include **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT** and **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT**.
Note

The `formatFeatures` member only indicates the features available when using an **external-format image** created from the Android hardware buffer. Images from Android hardware buffers with a format other than `VK_FORMAT_UNDEFINED` are subject to the format capabilities obtained from `vkGetPhysicalDeviceFormatProperties2` and `vkGetPhysicalDeviceImageFormatProperties2` with appropriate parameters. These sets of features are independent of each other, e.g. the external format will support sampler YCbCr conversion even if the non-external format does not, and writing to non-external format images is possible but writing to external format images is not.

Android hardware buffers with the same external format must have the same support for `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`, `VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT`, `VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT`, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT`, `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT`, and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT` in `formatFeatures`. Other format features may differ between Android hardware buffers that have the same external format. This allows applications to use the same `VkSamplerYcbcrConversion` object (and samplers and pipelines created from them) for any Android hardware buffers that have the same external format.

If `format` is not `VK_FORMAT_UNDEFINED`, then the value of `samplerYcbcrConversionComponents` must be valid when used as the `components` member of `VkSamplerYcbcrConversionCreateInfo` with that format. If `format` is `VK_FORMAT_UNDEFINED`, all members of `samplerYcbcrConversionComponents` must be the identity swizzle.

Implementations may not always be able to determine the color model, numerical range, or chroma offsets of the image contents, so the values in `VkAndroidHardwareBufferFormatPropertiesANDROID` are only suggestions. Applications should treat these values as sensible defaults to use in the absence of more reliable information obtained through some other means. If the underlying physical device is also usable via OpenGL ES with the `GL_OES_EGL_image_external` extension, the implementation should suggest values that will produce similar sampled values as would be obtained by sampling the same external image via `samplerExternalOES` in OpenGL ES using equivalent sampler parameters.

Note

Since `GL_OES_EGL_image_external` does not require the same sampling and conversion calculations as Vulkan does, achieving identical results between APIs may not be possible on some implementations.

**Valid Usage (Implicit)**

- `VUID-VkAndroidHardwareBufferFormatPropertiesANDROID-sType-sType` sType must be `VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_FORMAT_PROPERTIES_ANDROID`
When allocating memory that **may** be exported to another process or Vulkan instance, add a `VkExportMemoryAllocateInfoNV` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure, specifying the handle types that **may** be exported.

The `VkExportMemoryAllocateInfoNV` structure is defined as:

```c
// Provided by VK_NV_external_memory
typedef struct VkExportMemoryAllocateInfoNV {
    VkStructureType                      sType;
    const void*                          pNext;
    VkExternalMemoryHandleTypeFlagsNV    handleTypes;
} VkExportMemoryAllocateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` specifying one or more memory handle types that **may** be exported. Multiple handle types **may** be requested for the same allocation as long as they are compatible, as reported by `vkGetPhysicalDeviceExternalImageFormatPropertiesNV`.

### Valid Usage (Implicit)

- VUID-VkExportMemoryAllocateInfoNV-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV`.

- VUID-VkExportMemoryAllocateInfoNV-handleTypes-parameter
  
  `handleTypes` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values.

When `VkExportMemoryAllocateInfoNV::handleTypes` includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV`, add a `VkExportMemoryWin32HandleInfoNV` structure to the `pNext` chain of the `VkExportMemoryAllocateInfoNV` structure to specify security attributes and access rights for the memory object's external handle.

The `VkExportMemoryWin32HandleInfoNV` structure is defined as:

```c
// Provided by VK_NV_external_memory_win32
typedef struct VkExportMemoryWin32HandleInfoNV {
    VkStructureType               sType;
    const void*                   pNext;
    const SECURITY_ATTRIBUTES*    pAttributes;
    DWORD                         dwAccess;
} VkExportMemoryWin32HandleInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
• **pAttributes** is a pointer to a Windows **SECURITY_ATTRIBUTES** structure specifying security attributes of the handle.

• **dwAccess** is a DWORD specifying access rights of the handle.

If this structure is not present, or if **pAttributes** is set to **NULL**, default security descriptor values will be used, and child processes created by the application will not inherit the handle, as described in the MSDN documentation for “Synchronization Object Security and Access Rights”. Further, if the structure is not present, the access rights will be

**DXGI_SHARED_RESOURCE_READ | DXGI_SHARED_RESOURCE_WRITE**

1


---

**Valid Usage (Implicit)**

• VUID-VkExportMemoryWin32HandleInfoNV-sType-sType
  
  sType must be **VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV**

• VUID-VkExportMemoryWin32HandleInfoNV-pAttributes-parameter
  
  If **pAttributes** is not **NULL**, **pAttributes** must be a valid pointer to a valid **SECURITY_ATTRIBUTES** value

To import memory created on the same physical device but outside of the current Vulkan instance, add a **VkImportMemoryWin32HandleInfoNV** structure to the **pNext** chain of the **VkMemoryAllocateInfo** structure, specifying a handle to and the type of the memory.

The **VkImportMemoryWin32HandleInfoNV** structure is defined as:

```c
// Provided by VK_NV_external_memory_win32
typedef struct VkImportMemoryWin32HandleInfoNV {
    VkStructureType                      sType;
    const void*                          pNext;
    VkExternalMemoryHandleTypeFlagsNV    handleType;
    HANDLE                               handle;
} VkImportMemoryWin32HandleInfoNV;
```

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **handleType** is **0** or a **VkExternalMemoryHandleTypeFlagBitsNV** value specifying the type of memory handle in **handle**.

• **handle** is a Windows **HANDLE** referring to the memory.

If **handleType** is **0**, this structure is ignored by consumers of the **VkMemoryAllocateInfo** structure it is chained from.
Valid Usage

- VUID-VkImportMemoryWin32HandleInfoNV-handleType-01327
  handleType must not have more than one bit set

- VUID-VkImportMemoryWin32HandleInfoNV-handle-01328
  handle must be a valid handle to memory, obtained as specified by handleType

Valid Usage (Implicit)

- VUID-VkImportMemoryWin32HandleInfoNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_NV

- VUID-VkImportMemoryWin32HandleInfoNV-handleType-parameter
  handleType must be a valid combination of VkExternalMemoryHandleTypeFlagBitsNV values

Bits which can be set in handleType are:

Possible values of VkImportMemoryWin32HandleInfoNV::handleType, specifying the type of an external memory handle, are:

```cpp
// Provided by VK_NV_external_memory_capabilities
typedef enum VkExternalMemoryHandleTypeFlagBitsNV {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_NV = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_BIT_NV = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_KMT_BIT_NV = 0x00000008,
} VkExternalMemoryHandleTypeFlagBitsNV;
```

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_NV** specifies a handle to memory returned by vkGetMemoryWin32HandleNV.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV** specifies a handle to memory returned by vkGetMemoryWin32HandleNV, or one duplicated from such a handle using DuplicateHandle().

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_BIT_NV** specifies a valid NT handle to memory returned by IDXGIResource1::CreateSharedHandle, or a handle duplicated from such a handle using DuplicateHandle().

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_KMT_BIT_NV** specifies a handle to memory returned by IDXGIResource::GetSharedHandle().

```cpp
// Provided by VK_NV_external_memory_capabilities
typedef VkFlags VkExternalMemoryHandleTypeFlagsNV;
```

VkExternalMemoryHandleTypeFlagsNV is a bitmask type for setting a mask of zero or more...
VkExternalMemoryHandleTypeFlagBitsNV.

To retrieve the handle corresponding to a device memory object created with VkExportMemoryAllocateInfoNV::handleTypes set to include VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV or VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_NV, call:

```c
// Provided by VK_NV_external_memory_win32
VkResult vkGetMemoryWin32HandleNV(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    VkExternalMemoryHandleTypeFlagsNV           handleType,
    HANDLE*                                     pHandle);
```

- **device** is the logical device that owns the memory.
- **memory** is the VkDeviceMemory object.
- **handleType** is a bitmask of VkExternalMemoryHandleTypeFlagBitsNV containing a single bit specifying the type of handle requested.
- **handle** is a pointer to a Windows HANDLE in which the handle is returned.

## Valid Usage

- VUID-vkGetMemoryWin32HandleNV-handleType-01326
  **handleType** must be a flag specified in VkExportMemoryAllocateInfoNV::handleTypes when allocating memory

## Valid Usage (Implicit)

- VUID-vkGetMemoryWin32HandleNV-device-parameter
  **device** must be a valid VkDevice handle

- VUID-vkGetMemoryWin32HandleNV-memory-parameter
  **memory** must be a valid VkDeviceMemory handle

- VUID-vkGetMemoryWin32HandleNV-handleType-parameter
  **handleType** must be a valid combination of VkExternalMemoryHandleTypeFlagBitsNV values

- VUID-vkGetMemoryWin32HandleNV-handleType-required bitmask
  **handleType** must not be 0

- VUID-vkGetMemoryWin32HandleNV-pHandle-parameter
  **pHandle** must be a valid pointer to a HANDLE value

- VUID-vkGetMemoryWin32HandleNV-memory-parent
  **memory** must have been created, allocated, or retrieved from **device**
### Return Codes

<table>
<thead>
<tr>
<th><strong>Success</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUCCESS</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Failure</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ERROR_TOO_MANYOBJECTS</td>
<td></td>
</tr>
<tr>
<td>VK_ERROR_OUT_OF_HOST_MEMORY</td>
<td></td>
</tr>
</tbody>
</table>

#### 11.2.1. Fuchsia External Memory

On Fuchsia, when allocating memory that may be imported from another device, process or Vulkan instance, add a `VkImportMemoryZirconHandleInfoFUCHSIA` structure to the `pNext` chain of the `VkMemoryAllocateInfo` structure.

External memory on Fuchsia is imported and exported using VMO handles of type `zx_handle_t`. VMO handles to external memory are canonically obtained from Fuchsia’s Sysmem service or from syscalls such as `zx_vmo_create()`. VMO handles for import can also be obtained by exporting them from another Vulkan instance as described in exporting fuchsia device memory.

Importing VMO handles to the Vulkan instance transfers ownership of the handle to the instance from the application. The application **must** not perform any operations on the handle after successful import.

Applications **can** import the same underlying memory into multiple instances of Vulkan, into the same instance from which it was exported, and multiple times into a given Vulkan instance. In all cases, each import operation **must** create a distinct `VkDeviceMemory` object.

**Importing Fuchsia External Memory**

The `VkImportMemoryZirconHandleInfoFUCHSIA` structure is defined as:

```c
// Provided by VK_FUCHSIA_external_memory
typedef struct VkImportMemoryZirconHandleInfoFUCHSIA {
    VkStructureType                      sType;
    const void*                          pNext;
    VkExternalMemoryHandleTypeFlagBits   handleType;
    zx_handle_t                          handle;
} VkImportMemoryZirconHandleInfoFUCHSIA;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleType` is the type of `handle`.
- `handle` is a `zx_handle_t` (Zircon) handle to the external memory.
Valid Usage

- **VUID-VkImportMemoryZirconHandleInfoFUCHSIA-handleType-04771**
  
  handleType must be **VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA**.

- **VUID-VkImportMemoryZirconHandleInfoFUCHSIA-handle-04772**
  
  handle must be a valid VMO handle.

Valid Usage (Implicit)

- **VUID-VkImportMemoryZirconHandleInfoFUCHSIA-sType-sType**
  
  sType must be **VK_STRUCTURE_TYPE_IMPORT_MEMORY_ZIRCON_HANDLE_INFO_FUCHSIA**

- **VUID-VkImportMemoryZirconHandleInfoFUCHSIA-handleType-parameter**
  
  If handleType is not 0, handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

To obtain the memoryTypeIndex for the VkMemoryAllocateInfo structure, call vkGetMemoryZirconHandlePropertiesFUCHSIA:

```c
// Provided by VK_FUCHSIA_external_memory
VkResult vkGetMemoryZirconHandlePropertiesFUCHSIA(
    VkDevice                                    device,
    VkExternalMemoryHandleTypeFlagBits          handleType,
    zx_handle_t                                 zirconHandle,
    VkMemoryZirconHandlePropertiesFUCHSIA*      pMemoryZirconHandleProperties);
```

- **device** is the VkDevice.

- **handleType** is the type of zirconHandle

- **zirconHandle** is a zx_handle_t (Zircon) handle to the external resource.

- **pMemoryZirconHandleProperties** is a pointer to a VkMemoryZirconHandlePropertiesFUCHSIA structure in which the result will be stored.

Valid Usage

- **VUID-vkGetMemoryZirconHandlePropertiesFUCHSIA-handleType-04773**
  
  handleType must be **VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA**.

- **VUID-vkGetMemoryZirconHandlePropertiesFUCHSIA-zirconHandle-04774**
  
  zirconHandle must reference a valid VMO.
Valid Usage (Implicit)

- VUID-vkGetMemoryZirconHandlePropertiesFUCHSIA-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetMemoryZirconHandlePropertiesFUCHSIA-handleType-parameter
  handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

- VUID-vkGetMemoryZirconHandlePropertiesFUCHSIA-pMemoryZirconHandleProperties-parameter
  pMemoryZirconHandleProperties must be a valid pointer to a VkMemoryZirconHandlePropertiesFUCHSIA structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_INVALID_EXTERNAL_HANDLE

The VkMemoryZirconHandlePropertiesFUCHSIA structure is defined as:

```c
// Provided by VK_FUCHSIA_external_memory
typedef struct VkMemoryZirconHandlePropertiesFUCHSIA {
    VkStructureType sType;
    void* pNext;
    uint32_t memoryTypeBits;
} VkMemoryZirconHandlePropertiesFUCHSIA;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `memoryTypeBits` a bitmask containing one bit set for every memory type which the specified handle can be imported as.

Valid Usage (Implicit)

- VUID-VkMemoryZirconHandlePropertiesFUCHSIA-sType-sType
  sType must be VK_STRUCTURE_TYPE_MEMORY_ZIRCON_HANDLE_PROPERTIES_FUCHSIA

- VUID-VkMemoryZirconHandlePropertiesFUCHSIA-pNext-pNext
  pNext must be NULL

With pMemoryZirconHandleProperties now successfully populated by vkGetMemoryZirconHandlePropertiesFUCHSIA, assign the VkMemoryAllocateInfo memoryTypeIndex field to a memory type which has a bit set in the
VkMemoryZirconHandlePropertiesFUCHSIA memoryTypeBits field.

Exporting Fuchsia Device Memory

Similar to importing, exporting a VMO handle from Vulkan transfers ownership of the handle from the Vulkan instance to the application. The application is responsible for closing the handle with `zx_handle_close()` when it is no longer in use.

To export device memory as a Zircon handle that can be used by another instance, device, or process, the handle to the `VkDeviceMemory` must be retrieved using `vkGetMemoryZirconHandleFUCHSIA`:

```c
// Provided by VK_FUCHSIA_external_memory
VkResult vkGetMemoryZirconHandleFUCHSIA(
    VkDevice                                    device,
    const VkMemoryGetZirconHandleInfoFUCHSIA*   pGetZirconHandleInfo,
    zx_handle_t*                                pZirconHandle);
```

- `device` is the `VkDevice`.
- `pGetZirconHandleInfo` is a pointer to a `VkMemoryGetZirconHandleInfoFUCHSIA` structure.
- `pZirconHandle` is a pointer to a `zx_handle_t` which holds the resulting Zircon handle.

### Valid Usage (Implicit)

- VUID-vkGetMemoryZirconHandleFUCHSIA-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkGetMemoryZirconHandleFUCHSIA-pGetZirconHandleInfo-parameter
  
  `pGetZirconHandleInfo` must be a valid pointer to a valid `VkMemoryGetZirconHandleInfoFUCHSIA` structure

- VUID-vkGetMemoryZirconHandleFUCHSIA-pZirconHandle-parameter
  
  `pZirconHandle` must be a valid pointer to a `zx_handle_t` value

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

`VkMemoryGetZirconHandleInfoFUCHSIA` is defined as:
typedef struct VkMemoryGetZirconHandleInfoFUCHSIA {
    VkStructureType sType;
    const void* pNext;
    VkDeviceMemory memory;
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_FLAG_BITS_FUCHSIA handleType;
} VkMemoryGetZirconHandleInfoFUCHSIA;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **memory** the **VkDeviceMemory** being exported.
- **handleType** is the type of the handle pointed to by **pZirconHandle** as retrieved by the call to **vkGetMemoryZirconHandleFUCHSIA**.

### Valid Usage

- VUID-VkMemoryGetZirconHandleInfoFUCHSIA-handleType-04775
  - **handleType** must be **VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA**.
- VUID-VkMemoryGetZirconHandleInfoFUCHSIA-handleType-04776
  - **handleType** must have been included in the **handleTypes** field of the **VkExportMemoryAllocateInfo** structure when the external memory was allocated.

### Valid Usage (Implicit)

- VUID-VkMemoryGetZirconHandleInfoFUCHSIA-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_MEMORY_GET_ZIRCON_HANDLE_INFO_FUCHSIA**
- VUID-VkMemoryGetZirconHandleInfoFUCHSIA-pNext-pNext
  - **pNext** must be **NULL**
- VUID-VkMemoryGetZirconHandleInfoFUCHSIA-memory-parameter
  - **memory** must be a valid **VkDeviceMemory** handle
- VUID-VkMemoryGetZirconHandleInfoFUCHSIA-handleType-parameter
  - **handleType** must be a valid **VkExternalMemoryHandleTypeFlagBits** value

With the result **pZirconHandle** now obtained, the memory properties for the handle can be retrieved using **vkGetMemoryZirconHandlePropertiesFUCHSIA** as documented above substituting the dereferenced, retrieved **pZirconHandle** in for the **zirconHandle** argument.

If the **pNext** chain of **VkMemoryAllocateInfo** includes a **VkMemoryAllocateFlagsInfo** structure, then that structure includes flags and a device mask controlling how many instances of the memory will be allocated.

The **VkMemoryAllocateFlagsInfo** structure is defined as:
typedef struct VkMemoryAllocateFlagsInfo {
    VkStructureType sType;
    const void* pNext;
    VkMemoryAllocateFlags flags;
    uint32_t deviceMask;
} VkMemoryAllocateFlagsInfo;

or the equivalent

typedef VkMemoryAllocateFlagsInfo VkMemoryAllocateFlagsInfoKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkMemoryAllocateFlagBits` controlling the allocation.
- **deviceMask** is a mask of physical devices in the logical device, indicating that memory **must** be allocated on each device in the mask, if `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set in `flags`.

If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is not set, the number of instances allocated depends on whether `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set in the memory heap. If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is set, then memory is allocated for every physical device in the logical device (as if `deviceMask` has bits set for all device indices). If `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT` is not set, then a single instance of memory is allocated (as if `deviceMask` is set to one).

On some implementations, allocations from a multi-instance heap **may** consume memory on all physical devices even if the `deviceMask` excludes some devices. If `VkPhysicalDeviceGroupProperties::subsetAllocation` is `VK_TRUE`, then memory is only consumed for the devices in the device mask.

**Note**
In practice, most allocations on a multi-instance heap will be allocated across all physical devices. Unicast allocation support is an optional optimization for a minority of allocations.

**Valid Usage**

- **VUID-VkMemoryAllocateFlagsInfo-deviceMask-00675**
  If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set, `deviceMask` **must** be a valid device mask
- **VUID-VkMemoryAllocateFlagsInfo-deviceMask-00676**
  If `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT` is set, `deviceMask` **must** not be zero
Valid Usage (Implicit)

- VUID-VkMemoryAllocateFlagsInfo-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO

- VUID-VkMemoryAllocateFlagsInfo-flags-parameter
  
  flags must be a valid combination of VkMemoryAllocateFlagBits values

Bits which can be set in VkMemoryAllocateFlagsInfo::flags, controlling device memory allocation, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkMemoryAllocateFlagBits {
    VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT = 0x00000001,
    VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT = 0x00000002,
    VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT = 0x00000004,
    // Provided by VK_KHR_device_group
    VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHR = VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT,
    // Provided by VK_KHR_buffer_device_address
    VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR = VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT,
    // Provided by VK_KHR_buffer_device_address
    VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR = VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT,
} VkMemoryAllocateFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_device_group
typedef VkMemoryAllocateFlagBits VkMemoryAllocateFlagBitsKHR;
```

- VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT specifies that memory will be allocated for the devices in VkMemoryAllocateFlagsInfo::deviceMask.

- VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT specifies that the memory can be attached to a buffer object created with the VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT bit set in usage, and that the memory handle can be used to retrieve an opaque address via vkGetDeviceMemoryOpaqueCaptureAddress.

- VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT specifies that the memory's address can be saved and reused on a subsequent run (e.g. for trace capture and replay), see VkBufferOpaqueCaptureAddressCreateInfo for more detail.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkMemoryAllocateFlags;
```

or the equivalent
VkMemoryAllocateFlags is a bitmask type for setting a mask of zero or more VkMemoryAllocateFlagBits.

To request a specific device address for a memory allocation, add a VkMemoryOpaqueCaptureAddressAllocateInfo structure to the pNext chain of the VkMemoryAllocateInfo structure. The VkMemoryOpaqueCaptureAddressAllocateInfo structure is defined as:

```c
typedef struct VkMemoryOpaqueCaptureAddressAllocateInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint64_t           opaqueCaptureAddress;
} VkMemoryOpaqueCaptureAddressAllocateInfo;
```

or the equivalent

```c
typedef VkMemoryOpaqueCaptureAddressAllocateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `opaqueCaptureAddress` is the opaque capture address requested for the memory allocation.

If `opaqueCaptureAddress` is zero, no specific address is requested.

If `opaqueCaptureAddress` is not zero, it **should** be an address retrieved from `vkGetDeviceMemoryOpaqueCaptureAddress` on an identically created memory allocation on the same implementation.

**Note**

In most cases, it is expected that a non-zero `opaqueAddress` is an address retrieved from `vkGetDeviceMemoryOpaqueCaptureAddress` on an identically created memory allocation. If this is not the case, it likely that `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS` errors will occur.

This is, however, not a strict requirement because trace capture/replay tools may need to adjust memory allocation parameters for imported memory.

If this structure is not present, it is as if `opaqueCaptureAddress` is zero.
Valid Usage (Implicit)

- VUID-VkMemoryOpaqueCaptureAddressAllocateInfo-sType-sType
  
  sType **must** be VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO

To free a memory object, call:

```c
// Provided by VK_VERSION_1_0
void vkFreeMemory(
    VkDevice device,
    VkDeviceMemory memory,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that owns the memory.
- **memory** is the **VkDeviceMemory** object to be freed.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

Before freeing a memory object, an application **must** ensure the memory object is no longer in use by the device—for example by command buffers in the **pending state**. Memory **can** be freed whilst still bound to resources, but those resources **must** not be used afterwards. Freeing a memory object releases the reference it held, if any, to its payload. If there are still any bound images or buffers, the memory object’s payload **may** not be immediately released by the implementation, but **must** be released by the time all bound images and buffers have been destroyed. Once all references to a payload are released, it is returned to the heap from which it was allocated.

How memory objects are bound to Images and Buffers is described in detail in the Resource Memory Association section.

If a memory object is mapped at the time it is freed, it is implicitly unmapped.

**Note**

As described **below**, host writes are not implicitly flushed when the memory object is unmapped, but the implementation **must** guarantee that writes that have not been flushed do not affect any other memory.

Valid Usage

- VUID-vkFreeMemory-memory-00677
  
  All submitted commands that refer to **memory** (via images or buffers) **must** have completed execution
Valid Usage (Implicit)

- VUID-vkFreeMemory-device-parameter
  \textit{device} \textbf{must} be a valid \texttt{VkDevice} handle

- VUID-vkFreeMemory-memory-parameter
  If \textit{memory} is not \texttt{VK_NULL_HANDLE}, \textit{memory} \textbf{must} be a valid \texttt{VkDeviceMemory} handle

- VUID-vkFreeMemory-pAllocator-parameter
  If \textit{pAllocator} is not \texttt{NULL}, \textit{pAllocator} \textbf{must} be a valid pointer to a valid \texttt{VkAllocationCallbacks} structure

- VUID-vkFreeMemory-memory-parent
  If \textit{memory} is a valid handle, it \textbf{must} have been created, allocated, or retrieved from \texttt{device}

Host Synchronization

- Host access to \textit{memory} \textbf{must} be externally synchronized

11.2.2. Host Access to Device Memory Objects

Memory objects created with \texttt{vkAllocateMemory} are not directly host accessible.

Memory objects created with the memory property \texttt{VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT} are considered \textit{mappable}. Memory objects \textbf{must} be mappable in order to be successfully mapped on the host.

To retrieve a host virtual address pointer to a region of a mappable memory object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkMapMemory(
    VkDevice device,  // Provided by VK_VERSION_1_0
    VkDeviceMemory memory,  // Provided by VK_VERSION_1_0
    VkDeviceSize offset,  // Provided by VK_VERSION_1_0
    VkDeviceSize size,  // Provided by VK_VERSION_1_0
    VkMemoryMapFlags flags,  // Provided by VK_VERSION_1_0
    void** ppData);  // Provided by VK_VERSION_1_0
```

- \texttt{device} is the logical device that owns the memory.
- \texttt{memory} is the \texttt{VkDeviceMemory} object to be mapped.
- \texttt{offset} is a zero-based byte offset from the beginning of the memory object.
- \texttt{size} is the size of the memory range to map, or \texttt{VK_WHOLE_SIZE} to map from \texttt{offset} to the end of the allocation.
- \texttt{flags} is reserved for future use.
- \texttt{ppData} is a pointer to a \texttt{void *} variable in which is returned a host-accessible pointer to the
beginning of the mapped range. This pointer minus offset must be aligned to at least 
\( \text{VkPhysicalDeviceLimits}::\text{minMemoryMapAlignment} \).

After a successful call to \texttt{vkMapMemory} the memory object \textit{memory} is considered to be currently \textit{host mapped}.

\begin{itemize}
  \item[Note] It is an application error to call \texttt{vkMapMemory} on a memory object that is already \textit{host mapped}.
  \item[Note] \texttt{vkMapMemory} will fail if the implementation is unable to allocate an appropriately sized contiguous virtual address range, e.g. due to virtual address space fragmentation or platform limits. In such cases, \texttt{vkMapMemory} must return \texttt{VK_ERROR_MEMORY_MAP_FAILED}. The application can improve the likelihood of success by reducing the size of the mapped range and/or removing unneeded mappings using \texttt{vkUnmapMemory}.
\end{itemize}

\texttt{vkMapMemory} does not check whether the device memory is currently in use before returning the host-accessible pointer. The application must guarantee that any previously submitted command that writes to this range has completed before the host reads from or writes to that range, and that any previously submitted command that reads from that range has completed before the host writes to that region (see here for details on fulfilling such a guarantee). If the device memory was allocated without the \texttt{VK_MEMORY_PROPERTY_HOST_COHERENT_BIT} set, these guarantees must be made for an extended range: the application must round down the start of the range to the nearest multiple of \texttt{VkPhysicalDeviceLimits::nonCoherentAtomSize}, and round the end of the range up to the nearest multiple of \texttt{VkPhysicalDeviceLimits::nonCoherentAtomSize}.

While a range of device memory is host mapped, the application is responsible for synchronizing both device and host access to that memory range.

\begin{itemize}
  \item[Note] It is important for the application developer to become meticulously familiar with all of the mechanisms described in the chapter on \textit{Synchronization and Cache Control} as they are crucial to maintaining memory access ordering.
\end{itemize}
Valid Usage

- VUID-vkMapMemory-memory-00678
  memory must not be currently host mapped

- VUID-vkMapMemory-offset-00679
  offset must be less than the size of memory

- VUID-vkMapMemory-size-00680
  If size is not equal to VK_WHOLE_SIZE, size must be greater than 0

- VUID-vkMapMemory-size-00681
  If size is not equal to VK_WHOLE_SIZE, size must be less than or equal to the size of the memory minus offset

- VUID-vkMapMemory-memory-00682
  memory must have been created with a memory type that reports VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT

- VUID-vkMapMemory-memory-00683
  memory must not have been allocated with multiple instances

Valid Usage (Implicit)

- VUID-vkMapMemory-device-parameter
  device must be a valid VkDevice handle

- VUID-vkMapMemory-memory-parameter
  memory must be a valid VkDeviceMemory handle

- VUID-vkMapMemory-flags-zerobitmask
  flags must be 0

- VUID-vkMapMemory-ppData-parameter
  pData must be a valid pointer to a pointer value

- VUID-vkMapMemory-memory-parent
  memory must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to memory must be externally synchronized
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_MEMORY_MAP_FAILED

// Provided by VK_VERSION_1_0
typedef VkFlags VkMemoryMapFlags;

VkMemoryMapFlags is a bitmask type for setting a mask, but is currently reserved for future use.

Two commands are provided to enable applications to work with non-coherent memory allocations: vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges.

Note
If the memory object was created with the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges are unnecessary and may have a performance cost. However, availability and visibility operations still need to be managed on the device. See the description of host access types for more information.

Note
While memory objects imported from a handle type of VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT or VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT are inherently mapped to host address space, they are not considered to be host mapped device memory unless they are explicitly host mapped using vkMapMemory. That means flushing or invalidating host caches with respect to host accesses performed on such memory through the original host pointer specified at import time is the responsibility of the application and must be performed with appropriate synchronization primitives provided by the platform which are outside the scope of Vulkan. vkFlushMappedMemoryRanges and vkInvalidateMappedMemoryRanges, however, can still be used on such memory objects to synchronize host accesses performed through the host pointer of the host mapped device memory range returned by vkMapMemory.

To flush ranges of non-coherent memory from the host caches, call:
// Provided by VK_VERSION_1_0
VkResult vkFlushMappedMemoryRanges(
    VkDevice device,                  // device is the logical device that owns the memory ranges.
    uint32_t memoryRangeCount,       // memoryRangeCount is the length of the pMemoryRanges array.
    const VkMappedMemoryRange* pMemoryRanges);  // pMemoryRanges is a pointer to an array of VkMappedMemoryRange structures describing the memory ranges to flush.

vkFlushMappedMemoryRanges guarantees that host writes to the memory ranges described by pMemoryRanges are made available to the host memory domain, such that they can be made available to the device memory domain via memory domain operations using the VK_ACCESS_HOST_WRITE_BIT access type.

Within each range described by pMemoryRanges, each set of nonCoherentAtomSize bytes in that range is flushed if any byte in that set has been written by the host since it was first host mapped, or the last time it was flushed. If pMemoryRanges includes sets of nonCoherentAtomSize bytes where no bytes have been written by the host, those bytes must not be flushed.

Unmapping non-coherent memory does not implicitly flush the host mapped memory, and host writes that have not been flushed may not ever be visible to the device. However, implementations must ensure that writes that have not been flushed do not become visible to any other memory.

Note

The above guarantee avoids a potential memory corruption in scenarios where host writes to a mapped memory object have not been flushed before the memory is unmapped (or freed), and the virtual address range is subsequently reused for a different mapping (or memory allocation).

Valid Usage (Implicit)

• VUID-vkFlushMappedMemoryRanges-device-parameter
  device must be a valid VkDevice handle

• VUID-vkFlushMappedMemoryRanges-pMemoryRanges-parameter
  pMemoryRanges must be a valid pointer to an array of memoryRangeCount valid VkMappedMemoryRange structures

• VUID-vkFlushMappedMemoryRanges-memoryRangeCount-arraylength
  memoryRangeCount must be greater than 0
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To invalidate ranges of non-coherent memory from the host caches, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkInvalidateMappedMemoryRanges(
    VkDevice                                    device,
    uint32_t                                    memoryRangeCount,
    const VkMappedMemoryRange*                  pMemoryRanges);
```

- device is the logical device that owns the memory ranges.
- memoryRangeCount is the length of the pMemoryRanges array.
- pMemoryRanges is a pointer to an array of VkMappedMemoryRange structures describing the memory ranges to invalidate.

vkInvalidateMappedMemoryRanges guarantees that device writes to the memory ranges described by pMemoryRanges, which have been made available to the host memory domain using the VK_ACCESS_HOST_WRITE_BIT and VK_ACCESS_HOST_READ_BIT access types, are made visible to the host. If a range of non-coherent memory is written by the host and then invalidated without first being flushed, its contents are undefined.

Within each range described by pMemoryRanges, each set of nonCoherentAtomSize bytes in that range is invalidated if any byte in that set has been written by the device since it was first host mapped, or the last time it was invalidated.

Note
Mapping non-coherent memory does not implicitly invalidate that memory.
Valid Usage (Implicit)

- VUID-vkInvalidateMappedMemoryRanges-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkInvalidateMappedMemoryRanges-pMemoryRanges-parameter
  
  **pMemoryRanges** must be a valid pointer to an array of **memoryRangeCount** valid **VkMappedMemoryRange** structures

- VUID-vkInvalidateMappedMemoryRanges-memoryRangeCount-arraylength
  
  **memoryRangeCount** must be greater than 0

Return Codes

Success

- **VK_SUCCESS**

Failure

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The **VkMappedMemoryRange** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMappedMemoryRange {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceMemory     memory;
    VkDeviceSize       offset;
    VkDeviceSize       size;
} VkMappedMemoryRange;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **memory** is the memory object to which this range belongs.
- **offset** is the zero-based byte offset from the beginning of the memory object.
- **size** is either the size of range, or **VK_WHOLE_SIZE** to affect the range from **offset** to the end of the current mapping of the allocation.
Valid Usage

- VUID-VkMappedMemoryRange-memory-00684
  - memory must be currently host mapped

- VUID-VkMappedMemoryRange-size-00685
  - If size is not equal to VK_WHOLE_SIZE, offset and size must specify a range contained within the currently mapped range of memory

- VUID-VkMappedMemoryRange-size-00686
  - If size is equal to VK_WHOLE_SIZE, offset must be within the currently mapped range of memory

- VUID-VkMappedMemoryRange-offset-00687
  - offset must be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize

- VUID-VkMappedMemoryRange-size-01389
  - If size is equal to VK_WHOLE_SIZE, the end of the current mapping of memory must either be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize bytes from the beginning of the memory object, or be equal to the end of the memory object

- VUID-VkMappedMemoryRange-size-01390
  - If size is not equal to VK_WHOLE_SIZE, size must either be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize, or offset plus size must equal the size of memory

Valid Usage (Implicit)

- VUID-VkMappedMemoryRange-sType-sType
  - sType must be VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE

- VUID-VkMappedMemoryRange-pNext-pNext
  - pNext must be NULL

- VUID-VkMappedMemoryRange-memory-parameter
  - memory must be a valid VkDeviceMemory handle

To unmap a memory object once host access to it is no longer needed by the application, call:

```c
// Provided by VK_VERSION_1_0
void vkUnmapMemory(
    VkDevice device,  // device is the logical device that owns the memory.
    VkDeviceMemory memory);  // memory is the memory object to be unmapped.
```
Valid Usage

- VUID-vkUnmapMemory-memory-00689
  memory must be currently host mapped

Valid Usage (Implicit)

- VUID-vkUnmapMemory-device-parameter
device must be a valid VkDevice handle
- VUID-vkUnmapMemory-memory-parameter
  memory must be a valid VkDeviceMemory handle
- VUID-vkUnmapMemory-memory-parent
  memory must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to memory must be externally synchronized

11.2.3. Lazily Allocated Memory

If the memory object is allocated from a heap with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set, that object's backing memory may be provided by the implementation lazily. The actual committed size of the memory may initially be as small as zero (or as large as the requested size), and monotonically increases as additional memory is needed.

A memory type with this flag set is only allowed to be bound to a VkImage whose usage flags include VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT.

Note

Using lazily allocated memory objects for framebuffer attachments that are not needed once a render pass instance has completed may allow some implementations to never allocate memory for such attachments.

To determine the amount of lazily-allocated memory that is currently committed for a memory object, call:

```
// Provided by VK_VERSION_1_0
void vkGetDeviceMemoryCommitment(
    VkDevice device,
    VkDeviceMemory memory,
    VkDeviceSize* pCommittedMemoryInBytes);
```

- device is the logical device that owns the memory.
• *memory* is the memory object being queried.
• *pCommittedMemoryInBytes* is a pointer to a *VkDeviceSize* value in which the number of bytes currently committed is returned, on success.

The implementation *may* update the commitment at any time, and the value returned by this query *may* be out of date.

The implementation guarantees to allocate any committed memory from the *heapIndex* indicated by the memory type that the memory object was created with.

### Valid Usage

- **VUID-vkGetDeviceMemoryCommitment-memory-00690**
  - *memory* must have been created with a memory type that reports *VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT*

### Valid Usage (Implicit)

- **VUID-vkGetDeviceMemoryCommitment-device-parameter**
  - *device* must be a valid *VkDevice* handle
- **VUID-vkGetDeviceMemoryCommitment-memory-parameter**
  - *memory* must be a valid *VkDeviceMemory* handle
- **VUID-vkGetDeviceMemoryCommitment-pCommittedMemoryInBytes-parameter**
  - *pCommittedMemoryInBytes* must be a valid pointer to a *VkDeviceSize* value
- **VUID-vkGetDeviceMemoryCommitment-memory-parent**
  - *memory* must have been created, allocated, or retrieved from *device*

### 11.2.4. Protected Memory

*Protected memory* divides device memory into protected device memory and unprotected device memory.

Protected memory adds the following concepts:

- **Memory:**
  - Unprotected device memory, which *can* be visible to the device and *can* be visible to the host
  - Protected device memory, which *can* be visible to the device but *must* not be visible to the host
- **Resources:**
  - Unprotected images and unprotected buffers, to which unprotected memory *can* be bound
  - Protected images and protected buffers, to which protected memory *can* be bound
- **Command buffers:**
Unprotected command buffers, which can be submitted to a device queue to execute unprotected queue operations

Protected command buffers, which can be submitted to a protected-capable device queue to execute protected queue operations

• Device queues:
  ◦ Unprotected device queues, to which unprotected command buffers can be submitted
  ◦ Protected-capable device queues, to which unprotected command buffers or protected command buffers can be submitted

• Queue submissions
  ◦ Unprotected queue submissions, through which unprotected command buffers can be submitted
  ◦ Protected queue submissions, through which protected command buffers can be submitted

• Queue operations
  ◦ Unprotected queue operations
  ◦ Protected queue operations

Protected Memory Access Rules

If VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault is VK_FALSE, applications must not perform any of the following operations:

• Write to unprotected memory within protected queue operations.
• Access protected memory within protected queue operations other than in framebuffer-space pipeline stages, the compute shader stage, or the transfer stage.
• Perform a query within protected queue operations.
• Execute an indirect command within protected queue operations.

If VkPhysicalDeviceProtectedMemoryProperties::protectedNoFault is VK_TRUE, these operations are valid, but reads will return undefined values, and writes will either be dropped or store undefined values.

Whether these operations are valid or not, or if any other invalid usage is performed, the implementation must guarantee that:

• Protected device memory must never be visible to the host.
• Values written to unprotected device memory must not be a function of values from protected memory.

11.2.5. External Memory Handle Types

Android Hardware Buffer

Android’s NDK defines AHardwareBuffer objects, which represent device memory that is shareable across processes and that can be accessed by a variety of media APIs and the hardware used to
implement them. These Android hardware buffer objects may be imported into VkDeviceMemory objects for access via Vulkan, or exported from Vulkan. An VkImage or VkBuffer can be bound to the imported or exported VkDeviceMemory object if it is created with VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID.

To remove an unnecessary compile-time dependency, an incomplete type definition of AHardwareBuffer is provided in the Vulkan headers:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
struct AHardwareBuffer;
```

The actual AHardwareBuffer type is defined in Android NDK headers.

**Note**
The NDK format, usage, and size/dimensions of an AHardwareBuffer object can be obtained with the AHardwareBuffer_describe function. While Android hardware buffers can be imported to or exported from Vulkan without using that function, valid usage and implementation behavior is defined in terms of the AHardwareBuffer_Desc properties it returns.

Android hardware buffer objects are reference-counted using Android NDK functions outside of the scope of this specification. A VkDeviceMemory imported from an Android hardware buffer or that can be exported to an Android hardware buffer must acquire a reference to its AHardwareBuffer object, and must release this reference when the device memory is freed. During the host execution of a Vulkan command that has an Android hardware buffer as a parameter (including indirect parameters viapNext chains), the application must not decrement the Android hardware buffer's reference count to zero.

Android hardware buffers can be mapped and unmapped for CPU access using the NDK functions. These lock and unlock APIs are considered to acquire and release ownership of the Android hardware buffer, and applications must follow the rules described in External Resource Sharing to transfer ownership between the Vulkan instance and these native APIs.

Android hardware buffers can be shared with external APIs and Vulkan instances on the same device, and also with foreign devices. When transferring ownership of the Android hardware buffer, the external and foreign special queue families described in Queue Family Ownership Transfer are not identical. All APIs which produce or consume Android hardware buffers are considered to use foreign devices, except OpenGL ES contexts and Vulkan logical devices that have matching device and driver UUIDs. Implementations may treat a transfer to or from the foreign queue family as if it were a transfer to or from the external queue family when the Android hardware buffer's usage only permits it to be used on the same physical device.

**Android Hardware Buffer Optimal Usages**

Vulkan buffer and image usage flags do not correspond exactly to Android hardware buffer usage flags. When allocating Android hardware buffers with non-Vulkan APIs, if any AHARDWAREBUFFER_USAGE_GPU_* usage bits are included, by default the allocator must allocate the memory in such a way that it supports Vulkan usages and creation flags in the usage equivalence
table which do not have Android hardware buffer equivalents.

An `VkAndroidHardwareBufferUsageANDROID` structure can be included in the `pNext` chain of a `VkImageFormatProperties2` instance passed to `vkGetPhysicalDeviceImageFormatProperties2` to obtain optimal Android hardware buffer usage flags for specific Vulkan resource creation parameters. Some usage flags returned by these commands are required based on the input parameters, but additional vendor-specific usage flags (`AHARDWAREBUFFER_USAGE_VENDOR_*`) may also be returned. Any Android hardware buffer allocated with these vendor-specific usage flags and imported to Vulkan must only be bound to resources created with parameters that are a subset of the parameters used to obtain the Android hardware buffer usage, since the memory may have been allocated in a way incompatible with other parameters. If an Android hardware buffer is successfully allocated with additional non-vendor-specific usage flags in addition to the recommended usage, it must support being used in the same ways as an Android hardware buffer allocated with only the recommended usage, and also in ways indicated by the additional usage.

**Android Hardware Buffer External Formats**

Android hardware buffers may represent images using implementation-specific formats, layouts, color models, etc., which do not have Vulkan equivalents. Such external formats are commonly used by external image sources such as video decoders or cameras. Vulkan can import Android hardware buffers that have external formats, but since the image contents are in an undiscoverable and possibly proprietary representation, images with external formats must only be used as sampled images, must only be sampled with a sampler that has Y’CbCr conversion enabled, and must have optimal tiling.

Images that will be backed by an Android hardware buffer can use an external format by setting `VkImageCreateInfo::format` to `VK_FORMAT_UNDEFINED` and including a `VkExternalFormatANDROID` structure in the `pNext` chain. Images can be created with an external format even if the Android hardware buffer has a format which has an equivalent Vulkan format to enable consistent handling of images from sources that might use either category of format. However, all images created with an external format are subject to the valid usage requirements associated with external formats, even if the Android hardware buffer's format has a Vulkan equivalent. The external format of an Android hardware buffer can be obtained by passing a `VkAndroidHardwareBufferFormatPropertiesANDROID` structure to `vkGetAndroidHardwareBufferPropertiesANDROID`.

**Android Hardware Buffer Image Resources**

Android hardware buffers have intrinsic width, height, format, and usage properties, so Vulkan images bound to memory imported from an Android hardware buffer must use dedicated allocations: `VkMemoryDedicatedRequirements::requiresDedicatedAllocation` must be `VK_TRUE` for images created with `VkExternalMemoryImageCreateInfo::handleTypes` that includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`. When creating an image that will be bound to an imported Android hardware buffer, the image creation parameters must be equivalent to the `AHardwareBuffer` properties as described by the valid usage of `VkMemoryAllocateInfo`. Similarly, device memory allocated for a dedicated image must not be exported to an Android hardware buffer until it has been bound to that image, and the implementation must return an Android hardware buffer with properties derived from the image:
• The width and height members of AHardwareBuffer_Desc must be the same as the width and height members of VkImageCreateInfo::extent, respectively.

• The layers member of AHardwareBuffer_Desc must be the same as the arrayLayers member of VkImageCreateInfo.

• The format member of AHardwareBuffer_Desc must be equivalent to VkImageCreateInfo::format as defined by AHardwareBuffer Format Equivalence.

• The usage member of AHardwareBuffer_Desc must include bits corresponding to bits included in VkImageCreateInfo::usage and VkImageCreateInfo::flags where such a correspondence exists according to AHardwareBuffer Usage Equivalence. It may also include additional usage bits, including vendor-specific usages. Presence of vendor usage bits may make the Android hardware buffer only usable in ways indicated by the image creation parameters, even when used outside Vulkan, in a similar way that allocating the Android hardware buffer with usage returned in VkAndroidHardwareBufferUsageANDROID does.

Implementations may support fewer combinations of image creation parameters for images with Android hardware buffer external handle type than for non-external images. Support for a given set of parameters can be determined by passing VkExternalImageFormatProperties to vkGetPhysicalDeviceImageFormatProperties2 with handleType set to VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID. Any Android hardware buffer successfully allocated outside Vulkan with usage that includes AHWAREBUFFER_USAGE_GPU_* must be supported when using equivalent Vulkan image parameters. If a given choice of image parameters are supported for import, they can also be used to create an image and memory that will be exported to an Android hardware buffer.

Table 14. AHardwareBuffer Format Equivalence

<table>
<thead>
<tr>
<th>AHardwareBuffer Format</th>
<th>Vulkan Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHWAREBUFFER_FORMAT_R8G8B8A8_UNORM</td>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_R8G8B8X8 UNORM 1</td>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_R8G8B8_UNORM</td>
<td>VK_FORMAT_R8G8B8_UNORM</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_R5G6B5_UNORM</td>
<td>VK_FORMAT_R5G6B5_UNORM_PACK16</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_R16G16B16A16 FLOAT</td>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_R10G10B10A2 UNORM</td>
<td>VK_FORMAT_A2B10G10R10 UNORM_PACK32</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_D16 UNORM</td>
<td>VK_FORMAT_D16_UNORM</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_D24 UNORM</td>
<td>VK_FORMAT_X8_D24 UNORM PACK32</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_D24 UNORM S8 UINT</td>
<td>VK_FORMAT_D24 UNORM S8 UINT</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_D32 FLOAT</td>
<td>VK_FORMAT_D32_FLOAT</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_D32 FLOAT S8 UINT</td>
<td>VK_FORMAT_D32 FLOAT S8 UINT</td>
</tr>
<tr>
<td>AHWAREBUFFER_FORMAT_S8 UINT</td>
<td>VK_FORMAT_S8 UINT</td>
</tr>
</tbody>
</table>

Table 15. AHardwareBuffer Usage Equivalence

<table>
<thead>
<tr>
<th>AHardwareBuffer Usage</th>
<th>Vulkan Usage or Creation Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>VK_IMAGE_USAGE_TRANSFER_SRC_BIT</td>
</tr>
<tr>
<td>None</td>
<td>VK_IMAGE_USAGE_TRANSFER_DST_BIT</td>
</tr>
<tr>
<td>AHardwareBuffer Usage</td>
<td>Vulkan Usage or Creation Flag</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_GPU_SAMPLED_IMAGE</td>
<td>VK_IMAGE_USAGE_SAMPLED_BIT</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_GPU_SAMPLED_IMAGE</td>
<td>VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_GPU_FRAMEBUFFER ³</td>
<td>VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_GPU_FRAMEBUFFER ³</td>
<td>VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_GPU_CUBE_MAP</td>
<td>VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_GPU_MIPMAP_COMPLETE</td>
<td>None²</td>
</tr>
<tr>
<td>AHWARDBUFFER_USAGE_PROTECTED_CONTENT</td>
<td>VK_IMAGE_CREATE_PROTECTED_BIT</td>
</tr>
<tr>
<td>None</td>
<td>VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT</td>
</tr>
<tr>
<td>None</td>
<td>VK_IMAGE_CREATE_EXTENDED_USAGE_BIT</td>
</tr>
</tbody>
</table>

1. Vulkan does not differentiate between `AHARDWAREBUFFER_FORMAT_R8G8B8A8_UNORM` and `AHARDWAREBUFFER_FORMAT_R8G8B8X8_UNORM`: they both behave as `VK_FORMAT_R8G8B8A8_UNORM`. After an external entity writes to a `AHARDWAREBUFFER_FORMAT_R8G8B8X8_UNORM` Android hardware buffer, the values read by Vulkan from the X/A channel are undefined. To emulate the traditional behavior of the X channel during sampling or blending, applications should use `VK_COMPONENT_SWIZZLE_ONE` in image view component mappings and `VK_BLEND_FACTOR_ONE` in color blend factors. There is no way to avoid copying these undefined values when copying from such an image to another image or buffer.

2. The `AHARDWAREBUFFER_USAGE_GPU_MIPMAP_COMPLETE` flag does not correspond to a Vulkan image usage or creation flag. Instead, its presence indicates that the Android hardware buffer contains a complete mipmap chain, and its absence indicates that the Android hardware buffer contains only a single mip level.

3. Only image usages valid for the format are valid. It would be invalid to take a Android Hardware Buffer with a format of `AHARDWAREBUFFER_FORMAT_R8G8B8A8_UNORM` that has a `AHARDWAREBUFFER_USAGE_GPU_FRAMEBUFFER` usage and try to create an image with `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`.

Note

When using `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` with Android hardware buffer images, applications should use `VkImageFormatListCreateInfo` to inform the implementation which view formats will be used with the image. For some common sets of format, this allows some implementations to provide significantly better performance when accessing the image via Vulkan.

Android Hardware Buffer Buffer Resources

Android hardware buffers with a format of `AHARDWAREBUFFER_FORMAT_BLOB` and usage that includes `AHARDWAREBUFFER_USAGE_GPU_DATA_BUFFER` can be used as the backing store for `VkBuffer` objects. Such Android hardware buffers have a size in bytes specified by their `width`; `height` and `layers` are both 1.
Unlike images, buffer resources backed by Android hardware buffers do not require dedicated allocations.

Exported AHardwareBuffer objects that do not have dedicated images must have a format of AHARDWAREBUFFER_FORMAT_BLOB, usage must include AHARDWAREBUFFER_USAGE_GPU_DATA_BUFFER, width must equal the device memory allocation size, and height and layers must be 1.

11.2.6. Peer Memory Features

Peer memory is memory that is allocated for a given physical device and then bound to a resource and accessed by a different physical device, in a logical device that represents multiple physical devices. Some ways of reading and writing peer memory may not be supported by a device.

To determine how peer memory can be accessed, call:

```c
// Provided by VK_VERSION_1_1
void vkGetDeviceGroupPeerMemoryFeatures(
    VkDevice device,
    uint32_t heapIndex,
    uint32_t localDeviceIndex,
    uint32_t remoteDeviceIndex,
    VkPeerMemoryFeatureFlags* pPeerMemoryFeatures);
```
or the equivalent command

```c
// Provided by VK_KHR_device_group
void vkGetDeviceGroupPeerMemoryFeaturesKHR(
    VkDevice device,
    uint32_t heapIndex,
    uint32_t localDeviceIndex,
    uint32_t remoteDeviceIndex,
    VkPeerMemoryFeatureFlags* pPeerMemoryFeatures);
```

- **device** is the logical device that owns the memory.
- **heapIndex** is the index of the memory heap from which the memory is allocated.
- **localDeviceIndex** is the device index of the physical device that performs the memory access.
- **remoteDeviceIndex** is the device index of the physical device that the memory is allocated for.
- **pPeerMemoryFeatures** is a pointer to a VkPeerMemoryFeatureFlags bitmask indicating which types of memory accesses are supported for the combination of heap, local, and remote devices.
Valid Usage

- VUID-vkGetDeviceGroupPeerMemoryFeatures-heapIndex-00691
  heapIndex must be less than memoryHeapCount
- VUID-vkGetDeviceGroupPeerMemoryFeatures-localDeviceIndex-00692
  localDeviceIndex must be a valid device index
- VUID-vkGetDeviceGroupPeerMemoryFeatures-remoteDeviceIndex-00693
  remoteDeviceIndex must be a valid device index
- VUID-vkGetDeviceGroupPeerMemoryFeatures-localDeviceIndex-00694
  localDeviceIndex must not equal remoteDeviceIndex

Valid Usage (Implicit)

- VUID-vkGetDeviceGroupPeerMemoryFeatures-device-parameter
  device must be a valid VkDevice handle
- VUID-vkGetDeviceGroupPeerMemoryFeatures-pPeerMemoryFeatures-parameter
  pPeerMemoryFeatures must be a valid pointer to a VkPeerMemoryFeatureFlags value

Bits which may be set in the value returned for `vkGetDeviceGroupPeerMemoryFeatures::pPeerMemoryFeatures`, indicating the supported peer memory features, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkPeerMemoryFeatureFlagBits {
    VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT = 0x00000001,
    VK_PEER_MEMORY_FEATURE_COPY_DST_BIT = 0x00000002,
    VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT = 0x00000004,
    VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT = 0x00000008,
    // Provided by VK_KHR_device_group
    VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT_KHR = VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT,
    // Provided by VK_KHR_device_group
    VK_PEER_MEMORY_FEATURE_COPY_DST_BIT_KHR = VK_PEER_MEMORY_FEATURE_COPY_DST_BIT,
    // Provided by VK_KHR_device_group
    VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT_KHR = VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT,
    // Provided by VK_KHR_device_group
    VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT_KHR = VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT,
} VkPeerMemoryFeatureFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_device_group
typedef VkPeerMemoryFeatureFlagBits VkPeerMemoryFeatureFlagBitsKHR;
```
• **VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT** specifies that the memory can be accessed as the source of any `vkCmdCopy*` command.

• **VK_PEER_MEMORY_FEATURE_COPY_DST_BIT** specifies that the memory can be accessed as the destination of any `vkCmdCopy*` command.

• **VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT** specifies that the memory can be read as any memory access type.

• **VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT** specifies that the memory can be written as any memory access type. Shader atomics are considered to be writes.

```
Note
The peer memory features of a memory heap also apply to any accesses that may be performed during image layout transitions.
```

**VK_PEER_MEMORY_FEATURE_COPY_DST_BIT** must be supported for all host local heaps and for at least one device local heap.

If a device does not support a peer memory feature, it is still valid to use a resource that includes both local and peer memory bindings with the corresponding access type as long as only the local bindings are actually accessed. For example, an application doing split-frame rendering would use framebuffer attachments that include both local and peer memory bindings, but would scissor the rendering to only update local memory.

```
// Provided by VK_VERSION_1_1
typedef VkFlags VkPeerMemoryFeatureFlags;
```

or the equivalent

```
// Provided by VK_KHR_device_group
typedef VkPeerMemoryFeatureFlags VkPeerMemoryFeatureFlagsKHR;
```

**VkPeerMemoryFeatureFlags** is a bitmask type for setting a mask of zero or more **VkPeerMemoryFeatureFlagBits**.

To query a 64-bit opaque capture address value from a memory object, call:

```
// Provided by VK_KHR_buffer_device_address
uint64_t vkGetDeviceMemoryOpaqueCaptureAddressKHR(
    VkDevice device,
    const VkDeviceMemoryOpaqueCaptureAddressInfo* pInfo);
```

• **device** is the logical device that the memory object was allocated on.

• **pInfo** is a pointer to a **VkDeviceMemoryOpaqueCaptureAddressInfo** structure specifying the memory object to retrieve an address for.
The 64-bit return value is an opaque address representing the start of pInfo->memory.

If the memory object was allocated with a non-zero value of VkMemoryOpaqueCaptureAddressAllocateInfo::opaqueCaptureAddress, the return value must be the same address.

Note
The expected usage for these opaque addresses is only for trace capture/replay tools to store these addresses in a trace and subsequently specify them during replay.

Valid Usage

• VUID-vkGetDeviceMemoryOpaqueCaptureAddress-None-03334
  The bufferDeviceAddress feature must be enabled

• VUID-vkGetDeviceMemoryOpaqueCaptureAddress-device-03335
  If device was created with multiple physical devices, then the bufferDeviceAddressMultiDevice feature must be enabled

Valid Usage (Implicit)

• VUID-vkGetDeviceMemoryOpaqueCaptureAddress-device-parameter
device must be a valid VkDevice handle

• VUID-vkGetDeviceMemoryOpaqueCaptureAddress-pInfo-parameter
  pInfo must be a valid pointer to a valid VkDeviceMemoryOpaqueCaptureAddressInfo structure

The VkDeviceMemoryOpaqueCaptureAddressInfo structure is defined as:

```c
typedef struct VkDeviceMemoryOpaqueCaptureAddressInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceMemory     memory;
} VkDeviceMemoryOpaqueCaptureAddressInfo;
```

or the equivalent

```c
// Provided by VK_KHR_buffer_device_address
typedef VkDeviceMemoryOpaqueCaptureAddressInfo
    VkDeviceMemoryOpaqueCaptureAddressInfoKHR;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.
• `memory` specifies the memory whose address is being queried.

### Valid Usage

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-memory-03336  
  `memory` must have been allocated with `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT`

### Valid Usage (Implicit)

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-sType-sType  
  `sType` must be `VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO`

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-pNext-pNext  
  `pNext` must be `NULL`

- VUID-VkDeviceMemoryOpaqueCaptureAddressInfo-memory-parameter  
  `memory` must be a valid `VkDeviceMemory` handle
Chapter 12. Resource Creation

Vulkan supports two primary resource types: buffers and images. Resources are views of memory with associated formatting and dimensionality. Buffers are essentially unformatted arrays of bytes whereas images contain format information, can be multidimensional and may have associated metadata.

12.1. Buffers

Buffers represent linear arrays of data which are used for various purposes by binding them to a graphics or compute pipeline via descriptor sets or via certain commands, or by directly specifying them as parameters to certain commands.

Buffers are represented by VkBuffer handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBuffer)
```

To create buffers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateBuffer(
    VkDevice device,
    const VkBufferCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkBuffer* pBuffer);
```

- `device` is the logical device that creates the buffer object.
- `pCreateInfo` is a pointer to a VkBufferCreateInfo structure containing parameters affecting creation of the buffer.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pBuffer` is a pointer to a VkBuffer handle in which the resulting buffer object is returned.

Valid Usage

- VUID-vkCreateBuffer-flags-00911
  If the flags member of pCreateInfo includes VK_BUFFER_CREATE_SPARSE_BINDING_BIT, creating this VkBuffer must not cause the total required sparse memory for all currently valid sparse resources on the device to exceed VkPhysicalDeviceLimits::sparseAddressSpaceSize
Valid Usage (Implicit)

- **VUID-vkCreateBuffer-device-parameter**
  - `device` must be a valid `VkDevice` handle

- **VUID-vkCreateBuffer-pCreateInfo-parameter**
  - `pCreateInfo` must be a valid pointer to a valid `VkBufferCreateInfo` structure

- **VUID-vkCreateBuffer-pAllocator-parameter**
  - If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateBuffer-pBuffer-parameter**
  - `pBuffer` must be a valid pointer to a `VkBuffer` handle

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR

The `VkBufferCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferCreateInfo {
    VkStructureType        sType;
    const void*            pNext;
    VkBufferCreateFlags    flags;
    VkDeviceSize           size;
    VkBufferUsageFlags     usage;
    VkSharingMode          sharingMode;
    uint32_t               queueFamilyIndexCount;
    const uint32_t*        pQueueFamilyIndices;
} VkBufferCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkBufferCreateFlagBits` specifying additional parameters of the buffer.
- `size` is the size in bytes of the buffer to be created.
- `usage` is a bitmask of `VkBufferUsageFlagBits` specifying allowed usages of the buffer.
• **sharingMode** is a [VkSharingMode](https://registry.vulkan.org/rpc/vulkan/structures/VkSharingMode) value specifying the sharing mode of the buffer when it will be accessed by multiple queue families.

• **queueFamilyIndexCount** is the number of entries in the `pQueueFamilyIndices` array.

• **pQueueFamilyIndices** is a list of queue families that will access this buffer (ignored if `sharingMode` is not `VK_SHARING_MODE_CONCURRENT`).
Valid Usage

- VUID-VkBufferCreateInfo-size-00912
  
  `size` must be greater than 0

- VUID-VkBufferCreateInfo-sharingMode-00913
  
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount` `uint32_t` values

- VUID-VkBufferCreateInfo-sharingMode-00914
  
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` must be greater than 1

- VUID-VkBufferCreateInfo-sharingMode-01419
  
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than `pQueueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device

- VUID-VkBufferCreateInfo-flags-00915
  
  If the `sparse bindings` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`

- VUID-VkBufferCreateInfo-flags-00916
  
  If the `sparse buffer residency` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`

- VUID-VkBufferCreateInfo-flags-00917
  
  If the `sparse aliased residency` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`

- VUID-VkBufferCreateInfo-flags-00918
  
  If `flags` contains `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`, it must also contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`

- VUID-VkBufferCreateInfo-pNext-00920
  
  If the `pNext` chain includes a `VkExternalMemoryBufferCreateInfo` structure, its `handleTypes` member must only contain bits that are also in `VkExternalBufferProperties::externalMemoryProperties.compatibleHandleTypes`, as returned by `vkGetPhysicalDeviceExternalBufferProperties` with `pExternalBufferInfo->handleType` equal to any one of the handle types specified in `VkExternalMemoryBufferCreateInfo::handleTypes`

- VUID-VkBufferCreateInfo-flags-01887
  
  If the protected memory feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_PROTECTED_BIT`

- VUID-VkBufferCreateInfo-None-01888
  
  If any of the bits `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` are set, `VK_BUFFER_CREATE_PROTECTED_BIT` must not also be set

- VUID-VkBufferCreateInfo-pNext-01571
If the `pNext` chain includes a `VkDedicatedAllocationBufferCreateInfoNV` structure, and the `dedicatedAllocation` member of the chained structure is `VK_TRUE`, then `flags` must not include `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`, `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`.

- **VUID-VkBufferCreateInfo-deviceAddress-02604**
  If `VkBufferDeviceAddressCreateInfoEXT::deviceAddress` is not zero, `flags` must include `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`

- **VUID-VkBufferCreateInfo-opaqueCaptureAddress-03337**
  If `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress` is not zero, `flags` must include `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`

- **VUID-VkBufferCreateInfo-flags-03338**
  If `flags` includes `VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, the `bufferDeviceAddressCaptureReplay` or `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT::bufferDeviceAddressCaptureReplay` feature must be enabled

- **VUID-VkBufferCreateInfo-usage-04813**
  If `usage` includes `VK_BUFFER_USAGE_VIDEO_DECODE_SRC_BIT_KHR`, `VK_BUFFER_USAGE_VIDEO_DECODE_DST_BIT_KHR`, then the `pNext` chain must include a valid `VkVideoProfilesKHR` structure which includes at least one `VkVideoProfileKHR` with a decode codec-operation.

- **VUID-VkBufferCreateInfo-usage-04814**
  If `usage` includes `VK_BUFFER_USAGE_VIDEO_ENCODE_SRC_BIT_KHR`, `VK_BUFFER_USAGE_VIDEO_ENCODE_DST_BIT_KHR`, then the `pNext` chain must include a valid `VkVideoProfilesKHR` structure which includes at least one `VkVideoProfileKHR` with an encode codec-operation.
Valid Usage (Implicit)

- **VUID-VkBufferCreateInfo-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO*

- **VUID-VkBufferCreateInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkBufferDeviceAddressCreateInfoEXT`, `VkBufferOpaqueCaptureAddressCreateInfo`, `VkDedicatedAllocationBufferCreateInfoNV`, `VkExternalMemoryBufferCreateInfo`, `VkVideoProfileKHR`, or `VkVideoProfilesKHR`.

- **VUID-VkBufferCreateInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkBufferCreateInfo-flags-parameter**
  
  *flags must be a valid combination of `VkBufferCreateFlagBits` values*

- **VUID-VkBufferCreateInfo-usage-parameter**
  
  *usage must be a valid combination of `VkBufferUsageFlagBits` values*

- **VUID-VkBufferCreateInfo-usage-required bitmask**
  
  *usage must not be 0*

- **VUID-VkBufferCreateInfo-sharingMode-parameter**
  
  *sharingMode must be a valid `VkSharingMode` value*

Bits which **can** be set in `VkBufferCreateInfo::usage`, specifying usage behavior of a buffer, are:
typedef enum VkBufferUsageFlagBits {
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_BUFFER_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000004,
    VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT = 0x00000008,
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT = 0x00000010,
    VK_BUFFER_USAGE_STORAGE_BUFFER_BIT = 0x00000020,
    VK_BUFFER_USAGE_INDEX_BUFFER_BIT = 0x00000040,
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT = 0x00000080,
    VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT = 0x00000100,
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT = 0x00020000,
    VK_BUFFER_USAGE_VIDEO_DECODE_SRC_BIT_KHR = 0x00002000,
    VK_BUFFER_USAGE_VIDEO_DECODE_DST_BIT_KHR = 0x00004000,
    VK_BUFFER_USAGE_VIDEO_ENCODE_DST_BIT_KHR = 0x00008000,
    VK_BUFFER_USAGE_VIDEO_ENCODE_SRC_BIT_KHR = 0x00010000,
    VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT = 0x00000800,
    VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT = 0x00001000,
    VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00000200,
    VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR = 0x00080000,
    VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR = 0x00100000,
    VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR = 0x00000400,
    VK_BUFFER_USAGE_RAY_TRACING_BIT_NV = VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR,
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_EXT = VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT,
    VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR = VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT,
} VkBufferUsageFlagBits;
• **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** specifies that the buffer can be used as the source of a transfer command (see the definition of **VK_PIPELINE_STAGE_TRANSFER_BIT**).

• **VK_BUFFER_USAGE_TRANSFER_DST_BIT** specifies that the buffer can be used as the destination of a transfer command.

• **VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT** specifies that the buffer can be used to create a VkBufferView suitable for occupying a VkDescriptorSet slot of type **VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER**.

• **VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT** specifies that the buffer can be used to create a VkBufferView suitable for occupying a VkDescriptorSet slot of type **VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER**.

• **VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT** specifies that the buffer can be used in a VkDescriptorBufferInfo suitable for occupying a VkDescriptorSet slot either of type **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER** or **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC**.

• **VK_BUFFER_USAGE_STORAGE_BUFFER_BIT** specifies that the buffer can be used in a VkDescriptorBufferInfo suitable for occupying a VkDescriptorSet slot either of type **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER** or **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC**.

• **VK_BUFFER_USAGE_INDEX_BUFFER_BIT** specifies that the buffer is suitable for passing as the buffer parameter to vkCmdBindIndexBuffer.

• **VK_BUFFER_USAGE_VERTEX_BUFFER_BIT** specifies that the buffer is suitable for passing as an element of the pBuffers array to vkCmdBindVertexBuffers.

• **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** specifies that the buffer is suitable for passing as the buffer parameter to vkCmdDrawIndirect, vkCmdDrawIndexedIndirect, vkCmdDrawMeshTasksIndirectNV, vkCmdDrawMeshTasksIndirectCountNV, or vkCmdDispatchIndirect. It is also suitable for passing as the buffer member of VkIndirectCommandsStreamNV, or sequencesCountBuffer or sequencesIndexBuffer or preprocessedBuffer member of VkGeneratedCommandsInfoNV.

• **VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT** specifies that the buffer is suitable for passing as the buffer parameter to vkCmdBeginConditionalRenderingEXT.

• **VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT** specifies that the buffer is suitable for using for binding as a transform feedback buffer with vkCmdBindTransformFeedbackBuffersEXT.

• **VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT** specifies that the buffer is suitable for using as a counter buffer with vkCmdBeginTransformFeedbackEXT and vkCmdEndTransformFeedbackEXT.

• **VK_BUFFER_USAGE_RAY_TRACING_BIT_NV** specifies that the buffer is suitable for use in vkCmdTraceRaysNV.

• **VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR** specifies that the buffer is suitable for use as a Shader Binding Table.

• **VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR** specifies that the buffer is suitable for use as a read-only input to an acceleration structure build.

• **VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR** specifies that the buffer is suitable for storage space for a VkAccelerationStructureKHR.

• **VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT** specifies that the buffer can be used to retrieve a
buffer device address via `vkGetBufferDeviceAddress` and use that address to access the buffer’s memory from a shader.

- `VK_BUFFER_USAGE_VIDEO_DECODE_SRC_BIT_KHR` specifies that the buffer can be used as the source bitstream buffer in a video decode operation.
- `VK_BUFFER_USAGE_VIDEO_DECODE_DST_BIT_KHR` specifies that the buffer can be used as the destination status buffer in a video decode operation.
- `VK_BUFFER_USAGE_VIDEO_ENCODE_DST_BIT_KHR` specifies that the buffer can be used as the destination bitstream buffer in a video encode operation.
- `VK_BUFFER_USAGE_VIDEO_ENCODE_DST_BIT_KHR` specifies that the buffer can be used as the destination status buffer in a video encode operation.

```cpp
// Provided by VK_VERSION_1_0
typedef VkFlags VkBufferUsageFlags;

VkBufferUsageFlags is a bitmask type for setting a mask of zero or more `VkBufferUsageFlagBits`.

Bits which can be set in `VkBufferCreateInfo::flags`, specifying additional parameters of a buffer, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkBufferCreateFlagBits {
    VK_BUFFER_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_BUFFER_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
    // Provided by VK_VERSION_1_1
    VK_BUFFER_CREATE_PROTECTED_BIT = 0x00000008,
    VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT = 0x00000010,
    // Provided by VK_EXT_buffer_device_address
    VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_EXT = VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT,
    // Provided by VK_KHR_buffer_device_address
    VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR = VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT,
} VkBufferCreateFlagBits;
```

- `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` specifies that the buffer will be backed using sparse memory binding.
- `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` specifies that the buffer can be partially backed using sparse memory binding. Buffers created with this flag **must** also be created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` flag.
- `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT` specifies that the buffer will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another buffer (or another portion of the same buffer). Buffers created with this flag **must** also be created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` flag.

---

730 | Chapter 12. Resource Creation
• **VK_BUFFER_CREATE_PROTECTED_BIT** specifies that the buffer is a protected buffer.

• **VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT** specifies that the buffer’s address *can* be saved and reused on a subsequent run (e.g. for trace capture and replay), see `VkBufferOpaqueCaptureAddressCreateInfo` for more detail.

See Sparse Resource Features and Physical Device Features for details of the sparse memory features supported on a device.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkBufferCreateFlags;
```

**VkBufferCreateFlags** is a bitmask type for setting a mask of zero or more **VkBufferCreateFlagBits**.

If the `pNext` chain includes a `VkDedicatedAllocationBufferCreateInfoNV` structure, then that structure includes an enable controlling whether the buffer will have a dedicated memory allocation bound to it.

The **VkDedicatedAllocationBufferCreateInfoNV** structure is defined as:

```c
// Provided by VK_NV_dedicated_allocation
typedef struct VkDedicatedAllocationBufferCreateInfoNV {
    VkStructureType    sType;
    const void*        pNext;
    VkBool32           dedicatedAllocation;
} VkDedicatedAllocationBufferCreateInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **dedicatedAllocation** specifies whether the buffer will have a dedicated allocation bound to it.

**Valid Usage (Implicit)**

```c
• VUID-VkDedicatedAllocationBufferCreateInfoNV-sType-sType
    sType must be VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_BUFFER_CREATE_INFO_NV
```

To define a set of external memory handle types that *may* be used as backing store for a buffer, add a **VkExternalMemoryBufferCreateInfo** structure to the `pNext` chain of the **VkBufferCreateInfo** structure. The **VkExternalMemoryBufferCreateInfo** structure is defined as:
```c
typedef struct VkExternalMemoryBufferCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkExternalMemoryHandleTypeFlags handleTypes;
} VkExternalMemoryBufferCreateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_memory
typedef VkExternalMemoryBufferCreateInfo VkExternalMemoryBufferCreateInfoKHR;
```

Note

A `VkExternalMemoryBufferCreateInfo` structure with a non-zero `handleTypes` field must be included in the creation parameters for a buffer that will be bound to memory that is either exported or imported.

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **handleTypes** is zero, or a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more external memory handle types.

### Valid Usage (Implicit)

- VUID-VkExternalMemoryBufferCreateInfo-sType-sType
  **sType** must be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO`

- VUID-VkExternalMemoryBufferCreateInfo-handleTypes-parameter
  **handleTypes** must be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values

To request a specific device address for a buffer, add a `VkBufferOpaqueCaptureAddressCreateInfo` structure to the `pNext` chain of the `VkBufferCreateInfo` structure. The `VkBufferOpaqueCaptureAddressCreateInfo` structure is defined as:

```c
typedef struct VkBufferOpaqueCaptureAddressCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint64_t opaqueCaptureAddress;
} VkBufferOpaqueCaptureAddressCreateInfo;
```

or the equivalent
// Provided by VK_KHR_buffer_device_address
typedef VkBufferOpaqueCaptureAddressCreateInfo
VkBufferOpaqueCaptureAddressCreateInfoKHR;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **opaqueCaptureAddress** is the opaque capture address requested for the buffer.

If **opaqueCaptureAddress** is zero, no specific address is requested.

If **opaqueCaptureAddress** is not zero, then it **should** be an address retrieved from
**vkGetBufferOpaqueCaptureAddress** for an identically created buffer on the same implementation.

If this structure is not present, it is as if **opaqueCaptureAddress** is zero.

Apps **should** avoid creating buffers with app-provided addresses and implementation-provided
addresses in the same process, to reduce the likelihood of **VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS**
errors.

**Note**
The expected usage for this is that a trace capture/replay tool will add the
**VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT** flag to all buffers that use
**VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT**, and during capture will save the
queried opaque device addresses in the trace. During replay, the buffers will be
created specifying the original address so any address values stored in the trace
data will remain valid.

Implementations are expected to separate such buffers in the GPU address space
so normal allocations will avoid using these addresses. Apps/tools should avoid
mixing app-provided and implementation-provided addresses for buffers created
with **VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT**, to avoid address space
allocation conflicts.

**Valid Usage (Implicit)**

• VUID-VkBufferOpaqueCaptureAddressCreateInfo-sType-sType
  sType must be **VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO**

Alternatively, to request a specific device address for a buffer, add a
**VkBufferDeviceAddressCreateInfoEXT** structure to the **pNext** chain of the **VkBufferCreateInfo**
structure. The **VkBufferDeviceAddressCreateInfoEXT** structure is defined as:
typedef struct VkBufferDeviceAddressCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDeviceAddress deviceAddress;
} VkBufferDeviceAddressCreateInfoEXT;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **deviceAddress** is the device address requested for the buffer.

If **deviceAddress** is zero, no specific address is requested.

If **deviceAddress** is not zero, then it must be an address retrieved from an identically created buffer on the same implementation. The buffer must also be bound to an identically created VkDeviceMemory object.

If this structure is not present, it is as if **deviceAddress** is zero.

Apps should avoid creating buffers with app-provided addresses and implementation-provided addresses in the same process, to reduce the likelihood of **VK_ERROR_INVALID_DEVICE_ADDRESS_EXT** errors.

Valid Usage (Implicit)

• VUID-VkBufferDeviceAddressCreateInfoEXT-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_CREATE_INFO_EXT**

To destroy a buffer, call:

void vkDestroyBuffer(
    VkDevice device,
    VkBuffer buffer,
    const VkAllocationCallbacks* pAllocator);

• **device** is the logical device that destroys the buffer.
• **buffer** is the buffer to destroy.
• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
12.2. Buffer Views

A buffer view represents a contiguous range of a buffer and a specific format to be used to interpret the data. Buffer views are used to enable shaders to access buffer contents interpreted as formatted data. In order to create a valid buffer view, the buffer must have been created with at least one of the following usage flags:

- VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT
- VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT

Buffer views are represented by VkBufferView handles:
To create a buffer view, call:

```c
VKResult vkCreateBufferView(
    VkDevice device,
    const VkBufferViewCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkBufferView* pView);
```

- **device** is the logical device that creates the buffer view.
- **pCreateInfo** is a pointer to a `VkBufferViewCreateInfo` structure containing parameters to be used to create the buffer.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pView** is a pointer to a `VkBufferView` handle in which the resulting buffer view object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateBufferView-device-parameter**
  `device` must be a valid `VkDevice` handle

- **VUID-vkCreateBufferView-pCreateInfo-parameter**
  `pCreateInfo` must be a valid pointer to a valid `VkBufferViewCreateInfo` structure

- **VUID-vkCreateBufferView-pAllocator-parameter**
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateBufferView-pView-parameter**
  `pView` must be a valid pointer to a `VkBufferView` handle

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The `VkBufferViewCreateInfo` structure is defined as:
typedef struct VkBufferViewCreateInfo {
    VkStructureType            sType;
    const void*                pNext;
    VkBufferViewCreateFlags    flags;
    VkBuffer                   buffer;
    VkFormat                   format;
    VkDeviceSize               offset;
    VkDeviceSize               range;
} VkBufferViewCreateInfo;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `buffer` is a `VkBuffer` on which the view will be created.
- `format` is a `VkFormat` describing the format of the data elements in the buffer.
- `offset` is an offset in bytes from the base address of the buffer. Accesses to the buffer view from shaders use addressing that is relative to this starting offset.
- `range` is a size in bytes of the buffer view. If `range` is equal to `VK_WHOLE_SIZE`, the range from `offset` to the end of the buffer is used. If `VK_WHOLE_SIZE` is used and the remaining size of the buffer is not a multiple of the texel block size of `format`, the nearest smaller multiple is used.
Valid Usage

- **VUID-VkBufferViewCreateInfo-offset-00925**
  
  *offset must be less than the size of buffer*

- **VUID-VkBufferViewCreateInfo-range-00928**
  
  If *range* is not equal to *VK_WHOLE_SIZE*, *range must be greater than 0*

- **VUID-VkBufferViewCreateInfo-range-00929**
  
  If *range* is not equal to *VK_WHOLE_SIZE*, *range must be an integer multiple of the texel block size of format*

- **VUID-VkBufferViewCreateInfo-range-00930**
  
  If *range* is not equal to *VK_WHOLE_SIZE*, the number of texel buffer elements given by \( \left\lfloor \frac{\text{range}}{\text{texel block size}} \right\rfloor \times \text{texels per block} \) where texel block size and texels per block are as defined in the Compatible Formats table for *format*, *must* be less than or equal to *VkPhysicalDeviceLimits::maxTexelBufferElements*

- **VUID-VkBufferViewCreateInfo-offset-00931**
  
  If *range* is not equal to *VK_WHOLE_SIZE*, the sum of *offset* and *range must be less than or equal to the size of buffer*

- **VUID-VkBufferViewCreateInfo-range-04059**
  
  If *range* is equal to *VK_WHOLE_SIZE*, the number of texel buffer elements given by \( \left\lfloor \frac{\text{(size} - \text{offset})}{\text{texel block size}} \right\rfloor \times \text{texels per block} \) where *size* is the size of *buffer*, and texel block size and texels per block are as defined in the Compatible Formats table for *format*, *must* be less than or equal to *VkPhysicalDeviceLimits::maxTexelBufferElements*

- **VUID-VkBufferViewCreateInfo-buffer-00932**
  
  *buffer must have been created with a usage value containing at least one of VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT*

- **VUID-VkBufferViewCreateInfo-buffer-00933**
  
  If *buffer* was created with usage containing VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT, *format must be supported for uniform texel buffers, as specified by the VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT flag inVkFormatProperties::bufferFeatures returned by vkGetPhysicalDeviceFormatProperties*

- **VUID-VkBufferViewCreateInfo-buffer-00934**
  
  If *buffer* was created with usage containing VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, *format must be supported for storage texel buffers, as specified by the VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT flag in VkFormatProperties::bufferFeatures returned by vkGetPhysicalDeviceFormatProperties*

- **VUID-VkBufferViewCreateInfo-buffer-00935**
  
  If *buffer* is non-sparse then it *must be bound completely and contiguously to a single VkDeviceMemory object*

- **VUID-VkBufferViewCreateInfo-offset-02749**
  
  If the texelBufferAlignment feature is not enabled, *offset must be a multiple of VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment*

- **VUID-VkBufferViewCreateInfo-buffer-02750**
  
  If the texelBufferAlignment feature is enabled and if *buffer* was created with usage
containing `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, offset must be a multiple of the lesser of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::storageTexelBufferOffsetAlignmentBytes` or, if `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::storageTexelBufferOffsetSingleTexelAlignment` is `VK_TRUE`, the size of a texel of the requested format. If the size of a texel is a multiple of three bytes, then the size of a single component of format is used instead.

- **VUID-VkBufferViewCreateInfo-buffer-02751**

  If the texelBufferAlignment feature is enabled and if buffer was created with usage containing `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT`, offset must be a multiple of the lesser of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::uniformTexelBufferOffsetAlignmentBytes` or, if `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT::uniformTexelBufferOffsetSingleTexelAlignment` is `VK_TRUE`, the size of a texel of the requested format. If the size of a texel is a multiple of three bytes, then the size of a single component of format is used instead.

### Valid Usage (Implicit)

- **VUID-VkBufferViewCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO`

- **VUID-VkBufferViewCreateInfo-pNext-pNext**
  
  `pNext` must be `NULL`

- **VUID-VkBufferViewCreateInfo-flags-zero bitmask**
  
  `flags` must be `0`

- **VUID-VkBufferViewCreateInfo-buffer-parameter**
  
  `buffer` must be a valid `VkBuffer` handle

- **VUID-VkBufferViewCreateInfo-format-parameter**
  
  `format` must be a valid `VkFormat` value

---

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkBufferViewCreateFlags;
```

`VkBufferViewCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To destroy a buffer view, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyBufferView(
    VkDevice device, 
    VkBufferView bufferView, 
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the buffer view.
• `bufferView` is the buffer view to destroy.

• `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyBufferView-bufferView-00936**
  All submitted commands that refer to `bufferView` must have completed execution

- **VUID-vkDestroyBufferView-bufferView-00937**
  If `VkAllocationCallbacks` were provided when `bufferView` was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyBufferView-bufferView-00938**
  If no `VkAllocationCallbacks` were provided when `bufferView` was created, `pAllocator` must be `NULL`

### Valid Usage (Implicit)

- **VUID-vkDestroyBufferView-device-parameter**
  `device` must be a valid `VkDevice` handle

- **VUID-vkDestroyBufferView-bufferView-parameter**
  If `bufferView` is not `VK_NULL_HANDLE`, `bufferView` must be a valid `VkBufferView` handle

- **VUID-vkDestroyBufferView-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkDestroyBufferView-bufferView-parent**
  If `bufferView` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `bufferView` must be externally synchronized

### 12.3. Images

Images represent multidimensional - up to 3 - arrays of data which can be used for various purposes (e.g. attachments, textures), by binding them to a graphics or compute pipeline via descriptor sets, or by directly specifying them as parameters to certain commands.

Images are represented by `VkImage` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImage)
```
To create images, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateImage(
    VkDevice device,
    const VkImageCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkImage* pImage);
```

- **device** is the logical device that creates the image.
- **pCreateInfo** is a pointer to a `VkImageCreateInfo` structure containing parameters to be used to create the image.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pImage** is a pointer to a `VkImage` handle in which the resulting image object is returned.

### Valid Usage

- **VUID-vkCreateImage-flags-00939**
  If the `flags` member of `pCreateInfo` includes `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, creating this `VkImage` must not cause the total required sparse memory for all currently valid sparse resources on the device to exceed `VkPhysicalDeviceLimits::sparseAddressSpaceSize`

### Valid Usage (Implicit)

- **VUID-vkCreateImage-device-parameter**
  **device** must be a valid `VkDevice` handle

- **VUID-vkCreateImage-pCreateInfo-parameter**
  **pCreateInfo** must be a valid pointer to a valid `VkImageCreateInfo` structure

- **VUID-vkCreateImage-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateImage-pImage-parameter**
  **pImage** must be a valid pointer to a `VkImage` handle
**Return Codes**

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkImageCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageCreateFlags flags;
    VkImageType imageType;
    VkFormat format;
    VkExtent3D extent;
    uint32_t mipLevels;
    uint32_t arrayLayers;
    VkSampleCountFlagBits samples;
    VkImageTiling tiling;
    VkImageUsageFlags usage;
    VkSharingMode sharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
    VkImageLayout initialLayout;
} VkImageCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is a bitmask of `VkImageCreateFlagBits` describing additional parameters of the image.
- `imageType` is a `VkImageType` value specifying the basic dimensionality of the image. Layers in array textures do not count as a dimension for the purposes of the image type.
- `format` is a `VkFormat` describing the format and type of the texel blocks that will be contained in the image.
- `extent` is a `VkExtent3D` describing the number of data elements in each dimension of the base level.
- `mipLevels` describes the number of levels of detail available for minified sampling of the image.
- `arrayLayers` is the number of layers in the image.
- `samples` is a `VkSampleCountFlagBits` specifying the number of samples per texel.
- `tiling` is a `VkImageTiling` value specifying the tiling arrangement of the texel blocks in memory.
- **usage** is a bitmask of *VkImageUsageFlagBits* describing the intended usage of the image.
- **sharingMode** is a *VkSharingMode* value specifying the sharing mode of the image when it will be accessed by multiple queue families.
- **queueFamilyIndexCount** is the number of entries in the *pQueueFamilyIndices* array.
- **pQueueFamilyIndices** is a list of queue families that will access this image (ignored if **sharingMode** is not *VK_SHARING_MODE_CONCURRENT*).
- **initialLayout** is a *VkImageLayout* value specifying the initial *VkImageLayout* of all image subresources of the image. See Image Layouts.

Images created with **tiling** equal to **VK_IMAGE_TILING_LINEAR** have further restrictions on their limits and capabilities compared to images created with **tiling** equal to **VK_IMAGE_TILING_OPTIMAL**. Creation of images with **tiling** **VK_IMAGE_TILING_LINEAR** may not be supported unless other parameters meet all of the constraints:

- **imageType** is **VK_IMAGE_TYPE_2D**
- **format** is not a depth/stencil format
- **mipLevels** is 1
- **arrayLayers** is 1
- **samples** is **VK_SAMPLE_COUNT_1_BIT**
- **usage** only includes **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** and/or **VK_IMAGE_USAGE_TRANSFER_DST_BIT**

Images created with a **format** from one of those listed in Formats requiring sampler **Y’C_bC_R** conversion for **VK_IMAGE_ASPECT_COLOR_BIT** image views have further restrictions on their limits and capabilities compared to images created with other formats. Creation of images with a format requiring **Y’C_bC_R** conversion may not be supported unless other parameters meet all of the constraints:

- **imageType** is **VK_IMAGE_TYPE_2D**
- **mipLevels** is 1
- **arrayLayers** is 1
- **samples** is **VK_SAMPLE_COUNT_1_BIT**

Implementations may support additional limits and capabilities beyond those listed above.

To determine the set of valid usage bits for a given format, call **vkGetPhysicalDeviceFormatProperties**.

If the size of the resultant image would exceed **maxResourceSize**, then **vkCreateImage** must fail and return **VK_ERROR_OUT_OF_DEVICE_MEMORY**. This failure **may** occur even when all image creation parameters satisfy their valid usage requirements.
Note
For images created without `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a usage bit is valid if it is supported for the format the image is created with.

For images created with `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT` a usage bit is valid if it is supported for at least one of the formats a `VkImageView` created from the image can have (see Image Views for more detail).
Image Creation Limits

Valid values for some image creation parameters are limited by a numerical upper bound or by inclusion in a bitset. For example, `VkImageCreateInfo::arrayLayers` is limited by `imageCreateMaxArrayLayers`, defined below; and `VkImageCreateInfo::samples` is limited by `imageCreateSampleCounts`, also defined below.

Several limiting values are defined below, as well as assisting values from which the limiting values are derived. The limiting values are referenced by the relevant valid usage statements of `VkImageCreateInfo`.

- Let `uint64_t imageCreateDrmFormatModifiers[]` be the set of Linux DRM format modifiers that the resultant image may have.
  - If tiling is not `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `imageCreateDrmFormatModifiers` is empty.
  - If `VkImageCreateInfo::pNext` contains `VkImageDrmFormatModifierExplicitCreateInfoEXT`, then `imageCreateDrmFormatModifiers` contains exactly one modifier, `VkImageDrmFormatModifierExplicitCreateInfoEXT::drmFormatModifier`.
  - If `VkImageCreateInfo::pNext` contains `VkImageDrmFormatModifierListCreateInfoEXT`, then `imageCreateDrmFormatModifiers` contains the entire array `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`.

- Let `VkBool32 imageCreateMaybeLinear` indicate if the resultant image may be linear.
  - If tiling is `VK_IMAGE_TILING_LINEAR`, then `imageCreateMaybeLinear` is true.
  - If tiling is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateMaybeLinear` is false.
  - If tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `imageCreateMaybeLinear` is true if and only if `imageCreateDrmFormatModifiers` contains `DRM_FORMAT_MOD_LINEAR`.

- Let `VkFormatFeatureFlags imageCreateFormatFeatures` be the set of valid format features available during image creation.
  - If tiling is `VK_IMAGE_TILING_LINEAR`, then `imageCreateFormatFeatures` is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.
  - If tiling is `VK_IMAGE_TILING_OPTIMAL`, and if the `pNext` chain includes no `VkExternalFormatANDROID` structure with non-zero `externalFormat`, then `imageCreateFormatFeatures` is value of `VkFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.
  - If tiling is `VK_IMAGE_TILING_OPTIMAL`, and if the `pNext` chain includes a `VkExternalFormatANDROID` structure with non-zero `externalFormat`, then `imageCreateFormatFeatures` is the value of `VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures` obtained by calling `vkGetAndroidHardwareBufferPropertiesANDROID` with a matching `externalFormat`.
value.

- If tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the value of `imageCreateFormatFeatures` is found by calling `vkGetPhysicalDeviceFormatProperties2` with `VkImageFormatProperties::format` equal to `VkImageCreateInfo::format` and with `VkDrmFormatModifierPropertiesListEXT` chained into `VkImageFormatProperties2`; by collecting all members of the returned array `VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties` whose `drmFormatModifier` belongs to `imageCreateDrmFormatModifiers`; and by taking the bitwise intersection, over the collected array members, of `drmFormatModifierTilingFeatures`. (The resultant `imageCreateFormatFeatures` may be empty).

- Let `VkImageFormatProperties2 imageCreateImageFormatPropertiesList[]` be defined as follows.

  - If `VkImageCreateInfo::pNext` contains no `VkExternalFormatANDROID` structure with non-zero `externalFormat`, then `imageCreateImageFormatPropertiesList` is the list of structures obtained by calling `vkGetPhysicalDeviceImageFormatProperties2`, possibly multiple times, as follows:

    - The parameters `VkPhysicalDeviceImageFormatInfo2::format`, `imageType`, `tiling`, `usage`, and `flags` must be equal to those in `VkImageCreateInfo`.

    - If `VkImageCreateInfo::pNext` contains a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` is not 0, then `VkPhysicalDeviceImageFormatInfo2::pNext` must contain a `VkPhysicalDeviceExternalImageFormatInfo` structure whose `handleType` is not 0; and `vkGetPhysicalDeviceImageFormatProperties2` must be called for each handle type in `VkExternalMemoryImageCreateInfo::handleTypes`, successively setting `VkPhysicalDeviceExternalImageFormatInfo::handleType` on each call.

    - If `VkImageCreateInfo::pNext` contains no `VkExternalMemoryImageCreateInfo` structure, or contains a structure whose `handleTypes` is 0, then `VkPhysicalDeviceImageFormatInfo2::pNext` must either contain no `VkPhysicalDeviceExternalImageFormatInfo` structure, or contain a structure whose `handleType` is 0.

    - If tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `VkPhysicalDeviceImageFormatInfo2::pNext` must contain a `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` structure where `sharingMode` is equal to `VkImageCreateInfo::sharingMode`; and, if `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `queueFamilyIndexCount` and `pQueueFamilyIndices` must be equal to those in `VkImageCreateInfo`; and, if `flags` contains `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, then the `VkImageFormatListCreateInfo` structure included in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2::pNext` must be equivalent to the one included in the `pNext` chain of `VkImageCreateInfo`; and `vkGetPhysicalDeviceImageFormatProperties2` must be called for each modifier in `imageCreateDrmFormatModifiers`, successively setting `VkPhysicalDeviceImageDrmFormatModifierInfoEXT::drmFormatModifier` on each call.

    - If tiling is not `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `VkPhysicalDeviceImageFormatInfo2::pNext` must contain no
VkPhysicalDeviceImageDrmFormatModifierInfoEXT structure.

- If any call to `vkGetPhysicalDeviceImageFormatProperties2` returns an error, then `imageCreateImageFormatPropertiesList` is defined to be the empty list.
- If `VkImageCreateInfo::pNext` contains a `VkExternalFormatANDROID` structure with non-zero `externalFormat`, then `imageCreateImageFormatPropertiesList` contains a single element where:
  - `VkImageFormatProperties::maxMipLevels` is \( \lfloor \log_2(\max(\text{extent.width, extent.height, extent.depth})) \rfloor + 1 \).
  - `VkImageFormatProperties::maxArrayLayers` is `VkPhysicalDeviceLimits::maxImageArrayLayers`.
  - Each component of `VkImageFormatProperties::maxExtent` is `VkPhysicalDeviceLimits::maxImageDimension2D`.
  - `VkImageFormatProperties::sampleCounts` contains exactly `VK_SAMPLE_COUNT_1_BIT`.

- Let `uint32_t imageCreateMaxMipLevels` be the minimum value of `VkImageFormatProperties::maxMipLevels` in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.
- Let `uint32_t imageCreateMaxArrayLayers` be the minimum value of `VkImageFormatProperties::maxArrayLayers` in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.
- Let `VkExtent3D imageCreateMaxExtent` be the component-wise minimum over all `VkImageFormatProperties::maxExtent` values in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.
- Let `VkSampleCountFlags imageCreateSampleCounts` be the intersection of each `VkImageFormatProperties::sampleCounts` in `imageCreateImageFormatPropertiesList`. The value is undefined if `imageCreateImageFormatPropertiesList` is empty.
Valid Usage

- **VUID-VkImageCreateInfo-imageCreateMaxMipLevels-02251**
  Each of the following values (as described in Image Creation Limits) **must** not be undefined imageCreateMaxMipLevels, imageCreateMaxArrayLayers, imageCreateMaxExtent, and imageCreateSampleCounts

- **VUID-VkImageCreateInfo-sharingMode-00941**
  If sharingMode is VK_SHARING_MODE_CONCURRENT, pQueueFamilyIndices **must** be a valid pointer to an array of queueFamilyIndexCount uint32_t values

- **VUID-VkImageCreateInfo-sharingMode-00942**
  If sharingMode is VK_SHARING_MODE_CONCURRENT, queueFamilyIndexCount **must** be greater than 1

- **VUID-VkImageCreateInfo-sharingMode-01420**
  If sharingMode is VK_SHARING_MODE_CONCURRENT, each element of pQueueFamilyIndices **must** be unique and **must** be less than pQueueFamilyPropertyCount returned by either vkGetPhysicalDeviceQueueFamilyProperties or vkGetPhysicalDeviceQueueFamilyProperties2 for the physicalDevice that was used to create device

- **VUID-VkImageCreateInfo-pNext-01974**
  If the pNext chain includes a VkExternalFormatANDROID structure, and its externalFormat member is non-zero the format **must** be VK_FORMAT_UNDEFINED

- **VUID-VkImageCreateInfo-pNext-01975**
  If the pNext chain does not include a VkExternalFormatANDROID structure, or does and its externalFormat member is 0, the format **must** not be VK_FORMAT_UNDEFINED

- **VUID-VkImageCreateInfo-extent-00944**
  extent.width **must** be greater than 0

- **VUID-VkImageCreateInfo-extent-00945**
  extent.height **must** be greater than 0

- **VUID-VkImageCreateInfo-extent-00946**
  extent.depth **must** be greater than 0

- **VUID-VkImageCreateInfo-mipLevels-00947**
  mipLevels **must** be greater than 0

- **VUID-VkImageCreateInfo-arrayLayers-00948**
  arrayLayers **must** be greater than 0

- **VUID-VkImageCreateInfo-flags-00949**
  If flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, imageType **must** be VK_IMAGE_TYPE_2D

- **VUID-VkImageCreateInfo-flags-02557**
  If flags contains VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT, imageType **must** be VK_IMAGE_TYPE_2D

- **VUID-VkImageCreateInfo-flags-00950**
  If flags contains VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT, imageType **must** be VK_IMAGE_TYPE_3D

748 | Chapter 12. Resource Creation
extent.width must be less than or equal to imageCreateMaxExtent.width (as defined in Image Creation Limits)

extent.height must be less than or equal to imageCreateMaxExtent.height (as defined in Image Creation Limits)

extent.depth must be less than or equal to imageCreateMaxExtent.depth (as defined in Image Creation Limits)

extent.width must be less than or equal to imageCreateMaxExtent.width (as defined in Image Creation Limits)

extent.height must be less than or equal to imageCreateMaxExtent.height (as defined in Image Creation Limits)

extent.depth must be less than or equal to imageCreateMaxExtent.depth (as defined in Image Creation Limits)

If imageType is VK_IMAGE_TYPE_2D and flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, extent.width and extent.height must be equal and arrayLayers must be greater than or equal to 6

If imageType is VK_IMAGE_TYPE_1D, both extent.height and extent.depth must be 1

If imageType is VK_IMAGE_TYPE_2D, extent.depth must be 1

mipLevels must be less than or equal to the number of levels in the complete mipmap chain based on extent.width, extent.height, and extent.depth

mipLevels must be less than or equal to imageCreateMaxMipLevels (as defined in Image Creation Limits)

arrayLayers must be less than or equal to imageCreateMaxArrayLayers (as defined in Image Creation Limits)

If imageType is VK_IMAGE_TYPE_3D, arrayLayers must be 1

If samples is not VK_SAMPLE_COUNT_1_BIT, then imageType must be VK_IMAGE_TYPE_2D, flags must not contain VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, mipLevels must be equal to 1, and imageCreateMaybeLinear (as defined in Image Creation Limits) must be false,

If samples is not VK_SAMPLE_COUNT_1_BIT, usage must not contain VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT

If usage includes VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, then bits other than VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT must not be set

If usage includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, extent.width must be less than or equal to
- VUID-VkImageCreateInfo-usage-00965
  If `usage` includes `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, `extent.height` must be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`.

- VUID-VkImageCreateInfo-usage-02559
  If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `extent.width` must be less than or equal to \( \frac{\text{maxFramebufferWidth}}{\text{minFragmentDensityTeoclSize}} \).

- VUID-VkImageCreateInfo-usage-02560
  If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `extent.height` must be less than or equal to \( \frac{\text{maxFramebufferHeight}}{\text{minFragmentDensityTeoclSize}} \).

- VUID-VkImageCreateInfo-usage-00966
  If `usage` includes `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, `usage` must also contain at least one of `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`.

- VUID-VkImageCreateInfo-samples-02258
  `samples` must be a bit value that is set in `imageCreateSampleCounts` (as defined in Image Creation Limits).

- VUID-VkImageCreateInfo-usage-00968
  If the multisampled storage images feature is not enabled, and `usage` contains `VK_IMAGE_USAGE_STORAGE_BIT`, `samples` must be `VK_SAMPLE_COUNT_1_BIT`.

- VUID-VkImageCreateInfo-flags-00969
  If the sparse bindings feature is not enabled, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`.

- VUID-VkImageCreateInfo-flags-01924
  If the sparse aliased residency feature is not enabled, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`.

- VUID-VkImageCreateInfo-tiling-04121
  If `tiling` is `VK_IMAGE_TILING_LINEAR`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

- VUID-VkImageCreateInfo-imageType-00970
  If `imageType` is `VK_IMAGE_TYPE_1D`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

- VUID-VkImageCreateInfo-imageType-00971
  If the sparse residency for 2D images feature is not enabled, and `imageType` is `VK_IMAGE_TYPE_2D`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

- VUID-VkImageCreateInfo-imageType-00972
  If the sparse residency for 3D images feature is not enabled, and `imageType` is `VK_IMAGE_TYPE_3D`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

- VUID-VkImageCreateInfo-imageType-00973
  If the sparse residency for images with 2 samples feature is not enabled, `imageType` is `VK_IMAGE_TYPE_2D`, and `samples` is `VK_SAMPLE_COUNT_2_BIT`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
If the sparse residency for images with 4 samples feature is not enabled, imageType is VK_IMAGE_TYPE_2D, and samples is VK_SAMPLE_COUNT_4_BIT, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

If the sparse residency for images with 8 samples feature is not enabled, imageType is VK_IMAGE_TYPE_2D, and samples is VK_SAMPLE_COUNT_8_BIT, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

If the sparse residency for images with 16 samples feature is not enabled, imageType is VK_IMAGE_TYPE_2D, and samples is VK_SAMPLE_COUNT_16_BIT, flags must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

If flags contains VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT, it must also contain VK_IMAGE_CREATE_SPARSE_BINDING_BIT.

If any of the bits VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT are set, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT must not also be set.

If the protected memory feature is not enabled, flags must not contain VK_IMAGE_CREATE_PROTECTED_BIT.

If any of the bits VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT are set, VK_IMAGE_CREATE_PROTECTED_BIT must not also be set.

If the pNext chain includes a VkExternalMemoryImageCreateInfoNV structure, it must not contain a VkExternalMemoryImageCreateInfo structure.

If the pNext chain includes a VkExternalMemoryImageCreateInfo structure, its handleTypes member must only contain bits that are also in VkExternalImageFormatProperties::externalMemoryProperties.compatibleHandleTypes, as returned by vkGetPhysicalDeviceImageFormatProperties2 with format, imageType, tiling, usage, and flags equal to those in this structure, and with a VkPhysicalDeviceExternalImageFormatInfo structure included in the pNext chain, with a handleType equal to any one of the handle types specified in VkExternalMemoryImageCreateInfo::handleTypes.

If the pNext chain includes a VkExternalMemoryImageCreateInfoNV structure, its handleTypes member must only contain bits that are also in VkExternalImageFormatPropertiesNV::externalMemoryProperties.compatibleHandleTypes, as returned by vkGetPhysicalDeviceExternalImageFormatPropertiesNV with format, imageType, tiling, usage, and flags equal to those in this structure, and with externalHandleType equal to any one of the handle types specified in.
VkExternalMemoryImageCreateInfoNV::handleTypes

If the logical device was created with VkDeviceGroupDeviceCreateInfo::physicalDeviceCount equal to 1, flags must not contain VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT

If flags contains VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT, then mipLevels must be one, arrayLayers must be one, imageType must be VK_IMAGE_TYPE_2D, and imageCreateMaybeLinear (as defined in Image Creation Limits) must be false

If flags contains VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT, then format must be a block-compressed image format, an ETC compressed image format, or an ASTC compressed image format

If flags contains VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT, then flags must also contain VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT

initialLayout must be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED

If the pNext chain includes a VkExternalMemoryImageCreateInfo or VkExternalMemoryImageCreateInfoNV structure whose handleTypes member is not 0, initialLayout must be VK_IMAGE_LAYOUT_UNDEFINED

If the image format is one of those listed in Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views, then mipLevels must be 1

If the image format is one of those listed in Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views, samples must be VK_SAMPLE_COUNT_1_BIT

If the image format is one of those listed in Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views, imageType must be VK_IMAGE_TYPE_2D

If the image format is one of those listed in Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views, and the ycbcrImageArrays feature is not enabled, arrayLayers must be 1

If format is a multi-planar format, and if imageCreateFormatFeatures (as defined in Image Creation Limits) does not contain VK_FORMAT_FEATURE_DISJOINT_BIT, then flags must not contain VK_IMAGE_CREATE_DISJOINT_BIT

If format is not a multi-planar format, and flags does not include VK_IMAGE_CREATE_ALIAS_BIT, flags must not contain VK_IMAGE_CREATE_DISJOINT_BIT
If `format` has a \_422 or \_420 suffix, width must be a multiple of 2

• VUID-VkImageCreateInfo-format-04713

If `format` has a \_420 suffix, height must be a multiple of 2

• VUID-VkImageCreateInfo-tiling-02261

If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the `pNext` chain must include exactly one of `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT` structures

• VUID-VkImageCreateInfo-pNext-02262

If the `pNext` chain includes a `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT` structure, then `tiling` must be `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`

• VUID-VkImageCreateInfo-tiling-02353

If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and `flags` contains `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, then the `pNext` chain must include a `VkImageFormatListCreateInfo` structure with non-zero `viewFormatCount`

• VUID-VkImageCreateInfo-flags-01533

If `flags` contains `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` `format` must be a depth or depth/stencil format

• VUID-VkImageCreateInfo-pNext-02393

If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` member includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `imageType` must be `VK_IMAGE_TYPE_2D`

• VUID-VkImageCreateInfo-pNext-02394

If the `pNext` chain includes a `VkExternalMemoryImageCreateInfo` structure whose `handleTypes` member includes `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`, `mipLevels` must either be 1 or equal to the number of levels in the complete mipmap chain based on `extent.width`, `extent.height`, and `extent.depth`

• VUID-VkImageCreateInfo-pNext-02396

If the `pNext` chain includes a `VkExternalFormatANDROID` structure whose `externalFormat` member is not 0, `flags` must not include `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`

• VUID-VkImageCreateInfo-pNext-02397

If the `pNext` chain includes a `VkExternalFormatANDROID` structure whose `externalFormat` member is not 0, `usage` must not include any usages except `VK_IMAGE_USAGE_SAMPLED_BIT`

• VUID-VkImageCreateInfo-pNext-02398

If the `pNext` chain includes a `VkExternalFormatANDROID` structure whose `externalFormat` member is not 0, `tiling` must be `VK_IMAGE_TILING_OPTIMAL`

• VUID-VkImageCreateInfo-format-02795

If `format` is a depth-stencil format, `usage` includes `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` and the `pNext` chain includes a `VkImageStencilUsageCreateInfo` structure, then its `VkImageStencilUsageCreateInfo::stencilUsage` member must also include `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`

• VUID-VkImageCreateInfo-format-02796
If format is a depth-stencil format, usage does not include VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and the pNext chain includes a VkImageStencilUsageCreateInfo structure, then its VkImageStencilUsageCreateInfo::stencilUsage member must also not include VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT.

- VUID-VkImageCreateInfo-format-02797

If format is a depth-stencil format, usage includes VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT, and the pNext chain includes a VkImageStencilUsageCreateInfo structure, then its VkImageStencilUsageCreateInfo::stencilUsage member must also include VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT.

- VUID-VkImageCreateInfo-format-02798

If format is a depth-stencil format and the pNext chain includes a VkImageStencilUsageCreateInfo structure with its stencilUsage member including VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, extent.width must be less than or equal to VkPhysicalDeviceLimits::maxFramebufferWidth.

- VUID-VkImageCreateInfo-format-02537

If format is a depth-stencil format and the pNext chain includes a VkImageStencilUsageCreateInfo structure with its stencilUsage member including VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, extent.height must be less than or equal to VkPhysicalDeviceLimits::maxFramebufferHeight.

- VUID-VkImageCreateInfo-format-02538

If the multisampled storage images feature is not enabled, format is a depth-stencil format and the pNext chain includes a VkImageStencilUsageCreateInfo structure with its stencilUsage including VK_IMAGE_USAGE_STORAGE_BIT, samples must be VK_SAMPLE_COUNT_1_BIT.

- VUID-VkImageCreateInfo-flags-02050

If flags contains VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV, imageType must be VK_IMAGE_TYPE_2D or VK_IMAGE_TYPE_3D.

- VUID-VkImageCreateInfo-flags-02051

If flags contains VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV, it must not contain VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT and the format must not be a depth/stencil format.

- VUID-VkImageCreateInfo-flags-02052

If flags contains VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV and imageType is VK_IMAGE_TYPE_2D, extent.width and extent.height must be greater than 1.

- VUID-VkImageCreateInfo-flags-02053

If flags contains VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV and imageType is VK_IMAGE_TYPE_3D, extent.width, extent.height, and extent.depth must be greater than 1.

- VUID-VkImageCreateInfo-imageType-02082

If usage includes VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, imageType...
**must** be `VK_IMAGE_TYPE_2D`

- VUID-VkImageCreateInfo-samples-02083
  If `usage` includes `VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`, `samples` must be `VK_SAMPLE_COUNT_1_BIT`

- VUID-VkImageCreateInfo-tiling-02084
  If `usage` includes `VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV`, `tiling` must be `VK_IMAGE_TILING_OPTIMAL`

- VUID-VkImageCreateInfo-flags-02565
  If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `tiling` must be `VK_IMAGE_TILING_OPTIMAL`

- VUID-VkImageCreateInfo-flags-02566
  If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `imageType` must be `VK_IMAGE_TYPE_2D`

- VUID-VkImageCreateInfo-flags-02567
  If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `flags` must not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`

- VUID-VkImageCreateInfo-flags-02568
  If `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`, `mipLevels` must be `1`

- VUID-VkImageCreateInfo-imageView2DOn3DImage-04459
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::imageView2DOn3DImage` is `VK_FALSE`, `flags` must not contain `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`.

- VUID-VkImageCreateInfo-multisampleArrayImage-04460
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::multisampleArrayImage` is `VK_FALSE`, and `samples` is not `VK_SAMPLE_COUNT_1_BIT`, then `arrayLayers` must be `1`.

- VUID-VkImageCreateInfo-pNext-04737
  If a `VkImageFormatListCreateInfo` structure was included in the `pNext` chain and `VkImageFormatListCreateInfo::viewFormatCount` is not zero then all of the formats in `VkImageFormatListCreateInfo::pViewFormats` must be compatible with the `format` as described in the compatibility table.

- VUID-VkImageCreateInfo-flags-04738
  If `flags` does not contain `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` and the `pNext` chain include a `VkImageFormatListCreateInfo` structure then `VkImageFormatListCreateInfo::viewFormatCount` must be `0` or `1`

- VUID-VkImageCreateInfo-usage-04815
  If `usage` includes `VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR`, `VK_IMAGE_USAGE_VIDEO_DECODE_SRC_BIT_KHR`, `VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR`, then the `pNext` chain must include a valid `VkVideoProfilesKHR` structure which includes at least one `VkVideoProfileKHR` with a decode codec-operation.

- VUID-VkImageCreateInfo-usage-04816
  If `usage` includes `VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR`, `VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR`, `VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR`, then the `pNext` chain must include a valid `VkVideoProfilesKHR` structure which includes at least one `VkVideoProfileKHR` with an encode codec-operation.
Valid Usage (Implicit)

- **VUID-VkImageCreateInfo-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO*

- **VUID-VkImageCreateInfo-pNext-pNext**
  
  Each *pNext* member of any structure (including this one) in the *pNext* chain must be either NULL or a pointer to a valid instance of VkDedicatedAllocationImageCreateInfoNV, VkExternalFormatANDROID, VkExternalMemoryImageCreateInfo, VkImageDrmFormatModifierExplicitCreateInfoEXT, VkImageDrmFormatModifierListCreateInfoEXT, VkImageFormatListCreateInfo, VkImageStencilUsageCreateInfo, VkImageSwapchainCreateInfoKHR, VkVideoProfileKHR, or VkVideoProfilesKHR

- **VUID-VkImageCreateInfo-sType-unique**
  
  The *sType* value of each struct in the *pNext* chain must be unique

- **VUID-VkImageCreateInfo-flags-parameter**
  
  *flags must be a valid combination of VkImageCreateFlagBits values*

- **VUID-VkImageCreateInfo-imageType-parameter**
  
  *imageType must be a valid VkImageType value*

- **VUID-VkImageCreateInfo-format-parameter**
  
  *format must be a valid VkFormat value*

- **VUID-VkImageCreateInfo-samples-parameter**
  
  *samples must be a valid VkSampleCountFlagBits value*

- **VUID-VkImageCreateInfo-tiling-parameter**
  
  *tiling must be a valid VkImageTiling value*

- **VUID-VkImageCreateInfo-usage-parameter**
  
  *usage must be a valid combination of VkImageUsageFlagBits values*

- **VUID-VkImageCreateInfo-usage-requiredbitmask**
  
  *usage must not be 0*

- **VUID-VkImageCreateInfo-sharingMode-parameter**
  
  *sharingMode must be a valid VkSharingMode value*

- **VUID-VkImageCreateInfo-initialLayout-parameter**
  
  *initialLayout must be a valid VkImageLayout value*

---

The `VkImageStencilUsageCreateInfo` structure is defined as:

```c
typedef struct VkImageStencilUsageCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageUsageFlags stencilUsage;
} VkImageStencilUsageCreateInfo;
```
or the equivalent

```c
// Provided by VK_EXT_separate_stencil_usage
typedef VkImageStencilUsageCreateInfo VkImageStencilUsageCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `stencilUsage` is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the stencil aspect of the image.

If the `pNext` chain of `VkImageCreateInfo` includes a `VkImageStencilUsageCreateInfo` structure, then that structure includes the usage flags specific to the stencil aspect of the image for an image with a depth-stencil format.

This structure specifies image usages which only apply to the stencil aspect of a depth/stencil format image. When this structure is included in the `pNext` chain of `VkImageCreateInfo`, the stencil aspect of the image must only be used as specified by `stencilUsage`. When this structure is not included in the `pNext` chain of `VkImageCreateInfo`, the stencil aspect of an image must only be used as specified `VkImageCreateInfo::usage`. Use of other aspects of an image are unaffected by this structure.

This structure can also be included in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` to query additional capabilities specific to image creation parameter combinations including a separate set of usage flags for the stencil aspect of the image using `vkGetPhysicalDeviceImageFormatProperties2`. When this structure is not included in the `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` then the implicit value of `stencilUsage` matches that of `VkPhysicalDeviceImageFormatInfo2::usage`.

### Valid Usage

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-02539
  If `stencilUsage` includes `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`, it must not include bits other than `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` or `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`

### Valid Usage (Implicit)

- VUID-VkImageStencilUsageCreateInfo-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO`

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-parameter
  `stencilUsage` must be a valid combination of `VkImageUsageFlagBits` values

- VUID-VkImageStencilUsageCreateInfo-stencilUsage-requiredbitmask
  `stencilUsage` must not be 0
If the pNext chain includes a VkDedicatedAllocationImageCreateInfoNV structure, then that structure includes an enable controlling whether the image will have a dedicated memory allocation bound to it.

The VkDedicatedAllocationImageCreateInfoNV structure is defined as:

```c
// Provided by VK_NV_dedicated_allocation
typedef struct VkDedicatedAllocationImageCreateInfoNV {
    VkStructureType    sType;
    const void*        pNext;
    VkBool32           dedicatedAllocation;
} VkDedicatedAllocationImageCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `dedicatedAllocation` specifies whether the image will have a dedicated allocation bound to it.

**Note**
Using a dedicated allocation for color and depth/stencil attachments or other large images may improve performance on some devices.

**Valid Usage**

- VUID-VkDedicatedAllocationImageCreateInfoNV-dedicatedAllocation-00994
  If `dedicatedAllocation` is `VK_TRUE`, `VkImageCreateInfo::flags` must not include `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`, `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`, or `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`

**Valid Usage (Implicit)**

- VUID-VkDedicatedAllocationImageCreateInfoNV-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV`

To define a set of external memory handle types that may be used as backing store for an image, add a VkExternalMemoryImageCreateInfo structure to the pNext chain of the VkImageCreateInfo structure. The VkExternalMemoryImageCreateInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalMemoryImageCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkExternalMemoryHandleTypeFlags    handleTypes;
} VkExternalMemoryImageCreateInfo;
```
or the equivalent

```c
// Provided by VK_KHR_external_memory
typedef VkExternalMemoryImageCreateInfo VkExternalMemoryImageCreateInfoKHR;
```

**Note**

A `VkExternalMemoryImageCreateInfo` structure with a non-zero `handleType` field must be included in the creation parameters for an image that will be bound to memory that is either exported or imported.

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleTypes` is zero, or a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying one or more external memory handle types.

**Valid Usage (Implicit)**

- `VUID-VkExternalMemoryImageCreateInfo-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO`
- `VUID-VkExternalMemoryImageCreateInfo-handleTypes-parameter`  
  `handleTypes` must be a valid combination of `VkExternalMemoryHandleTypeFlagBits` values

If the `pNext` chain includes a `VkExternalMemoryImageCreateInfoNV` structure, then that structure defines a set of external memory handle types that may be used as backing store for the image.

The `VkExternalMemoryImageCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_external_memory
typedef struct VkExternalMemoryImageCreateInfoNV {
    VkStructureType                      sType;
    const void*                          pNext;
    VkExternalMemoryHandleTypeFlagsNV    handleTypes;
} VkExternalMemoryImageCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleTypes` is zero, or a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` specifying one or more external memory handle types.
Valid Usage (Implicit)

- **VUID-VkExternalMemoryImageCreateInfoNV-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV`

- **VUID-VkExternalMemoryImageCreateInfoNV-handleTypes-parameter**
  
  `handleTypes` **must** be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values

To create an image with an **external format**, add a `VkExternalFormatANDROID` structure in the `pNext` chain of `VkImageCreateInfo`. `VkExternalFormatANDROID` is defined as:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
typedef struct VkExternalFormatANDROID {
    VkStructureType sType;
    void* pNext;
    uint64_t externalFormat;
} VkExternalFormatANDROID;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `externalFormat` is an implementation-defined identifier for the external format

If `externalFormat` is zero, the effect is as if the `VkExternalFormatANDROID` structure was not present. Otherwise, the `image` will have the specified external format.

Valid Usage

- **VUID-VkExternalFormatANDROID-externalFormat-01894**

  `externalFormat` **must** be 0 or a value returned in the `externalFormat` member of `VkAndroidHardwareBufferFormatPropertiesANDROID` by an earlier call to `vkGetAndroidHardwareBufferPropertiesANDROID`

Valid Usage (Implicit)

- **VUID-VkExternalFormatANDROID-sType-sType**

  `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_FORMAT_ANDROID`

If the `pNext` chain of `VkImageCreateInfo` includes a `VkImageSwapchainCreateInfoKHR` structure, then that structure includes a swapchain handle indicating that the image will be bound to memory from that swapchain.

The `VkImageSwapchainCreateInfoKHR` structure is defined as:
typedef struct VkImageSwapchainCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSwapchainKHR swapchain;
} VkImageSwapchainCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **swapchain** is **VK_NULL_HANDLE** or a handle of a swapchain that the image will be bound to.

### Valid Usage

- **VUID-VkImageSwapchainCreateInfoKHR-swapchain-00995**
  If **swapchain** is not **VK_NULL_HANDLE**, the fields of **VkImageCreateInfo** must match the implied image creation parameters of the swapchain

### Valid Usage (Implicit)

- **VUID-VkImageSwapchainCreateInfoKHR-sType-sType**
  **sType** must be **VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR**

- **VUID-VkImageSwapchainCreateInfoKHR-swapchain-parameter**
  If **swapchain** is not **VK_NULL_HANDLE**, **swapchain** must be a valid **VkSwapchainKHR** handle

If the **pNext** list of **VkImageCreateInfo** includes a **VkImageFormatListCreateInfo** structure, then that structure contains a list of all formats that can be used when creating views of this image.

The **VkImageFormatListCreateInfo** structure is defined as:

typedef struct VkImageFormatListCreateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t viewFormatCount;
    const VkFormat* pViewFormats;
} VkImageFormatListCreateInfo;

or the equivalent

// Provided by VK_KHR_image_format_list
typedef VkImageFormatListCreateInfo VkImageFormatListCreateInfoKHR;

Chapter 12. Resource Creation | 761
• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **viewFormatCount** is the number of entries in the **pViewFormats** array.

• **pViewFormats** is an array which lists of all formats which can be used when creating views of this image.

If **viewFormatCount** is zero, **pViewFormats** is ignored and the image is created as if the **VkImageFormatListCreateInfo** structure were not included in the **pNext** list of **VkImageCreateInfo**.

---

**Valid Usage (Implicit)**

- **VUID-VkImageFormatListCreateInfo-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO**

- **VUID-VkImageFormatListCreateInfo-pViewFormats-parameter**
  - If **viewFormatCount** is not 0, **pViewFormats** must be a valid pointer to an array of **viewFormatCount** valid **VkFormat** values

If the **pNext** chain of **VkImageCreateInfo** includes a **VkImageDrmFormatModifierListCreateInfoEXT** structure, then the image will be created with one of the Linux DRM format modifiers listed in the structure. The choice of modifier is implementation-dependent.

The **VkImageDrmFormatModifierListCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkImageDrmFormatModifierListCreateInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           drmFormatModifierCount;
    const uint64_t*    pDrmFormatModifiers;
} VkImageDrmFormatModifierListCreateInfoEXT;
```

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **drmFormatModifierCount** is the length of the **pDrmFormatModifiers** array.

• **pDrmFormatModifiers** is a pointer to an array of Linux DRM format modifiers.

---

**Valid Usage**

- **VUID-VkImageDrmFormatModifierListCreateInfoEXT-pDrmFormatModifiers-02263**
  - Each modifier in **pDrmFormatModifiers** must be compatible with the parameters in **VkImageCreateInfo** and its **pNext** chain, as determined by querying **VkPhysicalDeviceImageFormatInfo2** extended with **VkPhysicalDeviceImageDrmFormatModifierInfoEXT**
Valid Usage (Implicit)

• **VUID-VkImageDrmFormatModifierListCreateInfoEXT-sType-sType**
  
  *sType must be* `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT`

• **VUID-VkImageDrmFormatModifierListCreateInfoEXT-pDrmFormatModifiers-parameter**
  
  *pDrmFormatModifiers must be a valid pointer to an array of `drmFormatModifierCount` `uint64_t` values*

• **VUID-VkImageDrmFormatModifierListCreateInfoEXT-drmFormatModifierCount-arraylength**
  
  *`drmFormatModifierCount` must be greater than 0*

If the `pNext` chain of `VkImageCreateInfo` includes a `VkImageDrmFormatModifierExplicitCreateInfoEXT` structure, then the image will be created with the Linux DRM format modifier and memory layout defined by the structure.

The `VkImageDrmFormatModifierExplicitCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkImageDrmFormatModifierExplicitCreateInfoEXT {
    VkStructureType               sType;
    const void*                   pNext;
    uint64_t                      drmFormatModifier;
    uint32_t                      drmFormatModifierPlaneCount;
    const VkSubresourceLayout*    pPlaneLayouts;
} VkImageDrmFormatModifierExplicitCreateInfoEXT;
```

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **drmFormatModifier** is the Linux DRM format modifier with which the image will be created.

• **drmFormatModifierPlaneCount** is the number of memory planes in the image (as reported by `VkDrmFormatModifierPropertiesEXT`) as well as the length of the `pPlaneLayouts` array.

• **pPlaneLayouts** is a pointer to an array of `VkSubresourceLayout` structures describing the image's memory planes.

The `i`th member of `pPlaneLayouts` describes the layout of the image's `i`th memory plane (that is, `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT`). In each element of `pPlaneLayouts`, the implementation must ignore `size`. The implementation calculates the size of each plane, which the application can query with `vkGetImageSubresourceLayout`.

When creating an image with `VkImageDrmFormatModifierExplicitCreateInfoEXT`, it is the application's responsibility to satisfy all valid usage requirements. However, the implementation must validate that the provided `pPlaneLayouts`, when combined with the provided `drmFormatModifier` and other creation parameters in `VkImageCreateInfo` and its `pNext` chain, produce a valid image. (This validation is necessarily implementation-dependent and outside the scope of Vulkan, and therefore not described by valid usage requirements). If this validation fails, then `vkCreateImage` returns `VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT`.

Chapter 12. Resource Creation | 763
Valid Usage

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-drmFormatModifier-02264
drmFormatModifier must be compatible with the parameters in VkImageCreateInfo and its pNext chain, as determined by querying VkPhysicalDeviceImageFormatInfo2 extended with VkPhysicalDeviceImageDrmFormatModifierInfoEXT

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-drmFormatModifierPlaneCount-02265
drmFormatModifierPlaneCount must be equal to the VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount associated with VkImageCreateInfo::format and drmFormatModifier, as found by querying VkDrmFormatModifierPropertiesListEXT

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-size-02267
For each element of pPlaneLayouts, size must be 0

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-arrayPitch-02268
For each element of pPlaneLayouts, arrayPitch must be 0 if VkImageCreateInfo::arrayLayers is 1

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-depthPitch-02269
For each element of pPlaneLayouts, depthPitch must be 0 if VkImageCreateInfo::extent.depth is 1

Valid Usage (Implicit)

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-sType-sType
sType must be VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT

- VUID-VkImageDrmFormatModifierExplicitCreateInfoEXT-pPlaneLayouts-parameter
If drmFormatModifierPlaneCount is not 0, pPlaneLayouts must be a valid pointer to an array of drmFormatModifierPlaneCount VkSubresourceLayout structures

Bits which can be set in VkImageViewUsageCreateInfo::usage, or VkImageStencilUsageCreateInfo::stencilUsage, or VkImageCreateInfo::usage, specifying intended usage of an image, are:
typedef enum VkImageUsageFlagBits {
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_IMAGE_USAGE_SAMPLED_BIT = 0x00000004,
    VK_IMAGE_USAGE_STORAGE_BIT = 0x00000008,
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT = 0x00000010,
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000020,
    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT = 0x00000040,
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT = 0x00000080,
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_decode_queue
        VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR = 0x00000400,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_decode_queue
        VK_IMAGE_USAGE_VIDEO_DECODE_SRC_BIT_KHR = 0x00000800,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_decode_queue
        VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR = 0x00001000,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_encode_queue
        VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR = 0x00002000,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_encode_queue
        VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR = 0x00004000,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_encode_queue
        VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR = 0x00008000,
    #endif
    // Provided by VK_NV_shading_rate_image
    VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00000100,
    // Provided by VK_EXT_fragment_density_map
    VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT = 0x00000200,
    // Provided by VK_KHR_fragment_shading_rate
    VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR =
        VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV,
} VkImageUsageFlagBits;

- **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** specifies that the image can be used as the source of a transfer command.
- **VK_IMAGE_USAGE_TRANSFER_DST_BIT** specifies that the image can be used as the destination of a transfer command.
- **VK_IMAGE_USAGE_SAMPLED_BIT** specifies that the image can be used to create a VkImageView suitable
for occupying a VkDescriptorSet slot either of type VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and be sampled by a shader.

- VK_IMAGE_USAGE_STORAGE_BIT specifies that the image can be used to create a VkImageView suitable for occupying a VkDescriptorSet slot of type VK_DESCRIPTOR_TYPE_STORAGE_IMAGE.

- VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT specifies that the image can be used to create a VkImageView suitable for use as a color or resolve attachment in a VkFramebuffer.

- VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT specifies that the image can be used to create a VkImageView suitable for use as a depth/stencil or depth/stencil resolve attachment in a VkFramebuffer.

- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT specifies that the memory bound to this image will have been allocated with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT (see Memory Allocation for more detail). This bit can be set for any image that can be used to create a VkImageView suitable for use as a color, resolve, depth/stencil, or input attachment.

- VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT specifies that the image can be used to create a VkImageView suitable for occupying VkDescriptorSet slot of type VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT; be read from a shader as an input attachment; and be used as an input attachment in a framebuffer.

- VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT specifies that the image can be used to create a VkImageView suitable for use as a fragment density map image.

- VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR specifies that the image can be used to create a VkImageView suitable for use as a fragment shading rate attachment or shading rate image.

- VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR specifies that video decode operations can use the image as a reference picture, or a source of postprocessing, or transfer operation.

- VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR specifies that video encode operations can use the image to hold a reconstructed picture, and to act as a reference picture. This flag must be combined with VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR and VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR. VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR is required for Dpb image resources because these resources will be associated with reference pictures after acting as targets for video decode operations. VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR is required for Dpb image resources because the reference pictures associated with them act as information sources for subsequent decode operations.

- VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR specifies that the image can be used as an input picture for video encode operations.

- VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR specifies that video encode operations can use the image to hold a reconstructed picture, and to act as a reference picture. This flag must be combined with VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR and VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR. VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR is required for Dpb image resource because the reconstructed picture resulting from an encode operation will be written to it. VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR is required for Dpb image resources because it can be used as a reconstructed reference picture source for the encode operation.
typedef VkFlags VkImageUsageFlags;

VkImageUsageFlags is a bitmask type for setting a mask of zero or more VkImageUsageFlagBits.

When creating a VkImageView one of the following VkImageUsageFlagBits must be set:

- VK_IMAGE_USAGE_SAMPLED_BIT
- VK_IMAGE_USAGE_STORAGE_BIT
- VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
- VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
- VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
- VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR
- VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT

Bits which can be set in VkImageCreateInfo::flags, specifying additional parameters of an image, are:
typedef enum VkImageCreateFlagBits {
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_IMAGE_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
    VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT = 0x00000008,
    VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT = 0x00000010,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_ALIAS_BIT = 0x00000400,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT = 0x00000040,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT = 0x00000020,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT = 0x00000080,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_EXTENDED_USAGE_BIT = 0x00000100,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_PROTECTED_BIT = 0x00000800,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_CREATE_DISJOINT_BIT = 0x00000200,
    // Provided by VK_NV_corner_sampled_image
    VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV = 0x00002000,
    // Provided by VK_EXT_sample_locations
    VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT = 0x00001000,
    // Provided by VK_EXT_fragment_density_map
    VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT = 0x00004000,
    // Provided by VK_KHR_device_group with VK_KHR_bind_memory2
    VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR = VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT,
    // Provided by VK_KHR_maintenance1
    VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR = VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT,
    // Provided by VK_KHR_maintenance2
    VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT_KHR = VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT,
    // Provided by VK_KHR_maintenance2
    VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR = VK_IMAGE_CREATE_EXTENDED_USAGE_BIT,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_IMAGE_CREATE_DISJOINT_BIT_KHR = VK_IMAGE_CREATE_DISJOINT_BIT,
    // Provided by VK_KHR_bind_memory2
    VK_IMAGE_CREATE_ALIAS_BIT_KHR = VK_IMAGE_CREATE_ALIAS_BIT,
} VkImageCreateFlagBits;

- **VK_IMAGE_CREATE_SPARSE_BINDING_BIT** specifies that the image will be backed using sparse memory binding.
- **VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT** specifies that the image can be partially backed using sparse memory binding. Images created with this flag must also be created with the **VK_IMAGE_CREATE_SPARSE_BINDING_BIT** flag.
• **VK_IMAGE_CREATE_SPARSE_ALIASED_BIT** specifies that the image will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another image (or another portion of the same image). Images created with this flag must also be created with the **VK_IMAGE_CREATE_SPARSE_BINDING_BIT** flag.

• **VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT** specifies that the image can be used to create a VkImageView with a different format from the image. For multi-planar formats, **VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT** specifies that a VkImageView can be created of a plane of the image.

• **VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT** specifies that the image can be used to create a VkImageView of type **VK_IMAGE_VIEW_TYPE_CUBE** or **VK_IMAGE_VIEW_TYPE_CUBE_ARRAY**.

• **VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT** specifies that the image can be used to create a VkImageView of type **VK_IMAGE_VIEW_TYPE_2D** or **VK_IMAGE_VIEW_TYPE_2D_ARRAY**.

• **VK_IMAGE_CREATE_PROTECTED_BIT** specifies that the image is a protected image.

• **VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT** specifies that the image can be used with a non-zero value of the splitInstanceBindRegionCount member of a VkBindImageMemoryDeviceGroupInfo structure passed into vkBindImageMemory2. This flag also has the effect of making the image use the standard sparse image block dimensions.

• **VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT** specifies that the image having a compressed format can be used to create a VkImageView with an uncompressed format where each texel in the image view corresponds to a compressed texel block of the image.

• **VK_IMAGE_CREATE_EXTENDED_USAGE_BIT** specifies that the image can be created with usage flags that are not supported for the format the image is created with but are supported for at least one format a VkImageView created from the image can have.

• **VK_IMAGE_CREATE_DISJOINT_BIT** specifies that an image with a multi-planar format must have each plane separately bound to memory, rather than having a single memory binding for the whole image; the presence of this bit distinguishes a disjoint image from an image without this bit set.

• **VK_IMAGE_CREATE_ALIAS_BIT** specifies that two images created with the same creation parameters and aliased to the same memory can interpret the contents of the memory consistently with each other, subject to the rules described in the Memory Aliasing section. This flag further specifies that each plane of a disjoint image can share an in-memory non-linear representation with single-plane images, and that a single-plane image can share an in-memory non-linear representation with a plane of a multi-planar disjoint image, according to the rules in Compatible formats of planes of multi-planar formats. If the pNext chain includes a VkExternalMemoryImageCreateInfo or VkExternalMemoryImageCreateInfoNV structure whose handleTypes member is not 0, it is as if **VK_IMAGE_CREATE_ALIAS_BIT** is set.

• **VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT** specifies that an image with a depth or depth/stencil format can be used with custom sample locations when used as a depth/stencil attachment.

• **VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV** specifies that the image is a corner-sampled image.

• **VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT** specifies that an image can be in a subsampled format which may be more optimal when written as an attachment by a render pass that has a fragment density map attachment. Accessing a subsampled image has additional
considerations:

- Image data read as an image sampler will have undefined values if the sampler was not created with flags containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` or was not sampled through the use of a combined image sampler with an immutable sampler in `VkDescriptorSetLayoutBinding`.

- Image data read with an input attachment will have undefined values if the contents were not written as an attachment in an earlier subpass of the same render pass.

- Image data read as an image sampler in the fragment shader will be additionally be read by the device during `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT` if `VkPhysicalDeviceFragmentDensityMap2PropertiesEXT::subsampledCoarseReconstructionEarlyAccess` is `VK_TRUE` and the sampler was created with flags containing `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT`.

- Image data read with load operations are resampled to the fragment density of the render pass if `VkPhysicalDeviceFragmentDensityMap2PropertiesEXT::subsampledLoads` is `VK_TRUE`. Otherwise, values of image data are undefined.

- Image contents outside of the render area take on undefined values if the image is stored as a render pass attachment.

See Sparse Resource Features and Sparse Physical Device Features for more details.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkImageCreateFlags;
```

`VkImageCreateFlags` is a bitmask type for setting a mask of zero or more `VkImageCreateFlagBits`.

Possible values of `VkImageCreateInfo::imageType`, specifying the basic dimensionality of an image, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageType {
    VK_IMAGE_TYPE_1D = 0,
    VK_IMAGE_TYPE_2D = 1,
    VK_IMAGE_TYPE_3D = 2,
} VkImageType;
```

- `VK_IMAGE_TYPE_1D` specifies a one-dimensional image.
- `VK_IMAGE_TYPE_2D` specifies a two-dimensional image.
- `VK_IMAGE_TYPE_3D` specifies a three-dimensional image.

Possible values of `VkImageCreateInfo::tiling`, specifying the tiling arrangement of texel blocks in an image, are:
• **VK_IMAGE_TILING_OPTIMAL** specifies optimal tiling (texels are laid out in an implementation-dependent arrangement, for more optimal memory access).

• **VK_IMAGE_TILING_LINEAR** specifies linear tiling (texels are laid out in memory in row-major order, possibly with some padding on each row).

• **VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT** indicates that the image’s tiling is defined by a Linux DRM format modifier. The modifier is specified at image creation with `VkImageDrmFormatModifierListCreateInfoEXT` or `VkImageDrmFormatModifierExplicitCreateInfoEXT`, and can be queried with `vkGetImageDrmFormatModifierPropertiesEXT`.

To query the memory layout of an image subresource, call:

```c
// Provided by VK_VERSION_1_0
void vkGetImageSubresourceLayout(
    VkDevice device,
    VkImage image,
    const VkImageSubresource* pSubresource,
    VkSubresourceLayout* pLayout);
```

• **device** is the logical device that owns the image.

• **image** is the image whose layout is being queried.

• **pSubresource** is a pointer to a `VkImageSubresource` structure selecting a specific image for the image subresource.

• **pLayout** is a pointer to a `VkSubresourceLayout` structure in which the layout is returned.

If the image is **linear**, then the returned layout is valid for **host access**.

If the image’s tiling is **VK_IMAGE_TILING_LINEAR** and its format is a **multi-planar format**, then `vkGetImageSubresourceLayout` describes one **format plane** of the image. If the image’s tiling is **VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT**, then `vkGetImageSubresourceLayout` describes one **memory plane** of the image. If the image’s tiling is **VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT** and the image is **non-linear**, then the returned layout has an implementation-dependent meaning; the vendor of the image’s DRM format modifier may provide documentation that explains how to interpret the returned layout.

`vkGetImageSubresourceLayout` is invariant for the lifetime of a single image. However, the subresource layout of images in Android hardware buffer external memory is not known until the
image has been bound to memory, so applications **must** not call `vkGetImageSubresourceLayout` for such an image before it has been bound.
Valid Usage

- **VUID-vkGetImageSubresourceLayout-image-02270**
  
  *image* must have been created with *tiling* equal to *VK_IMAGE_TILING_LINEAR* or *VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT*

- **VUID-vkGetImageSubresourceLayout-aspectMask-00997**
  
  The *aspectMask* member of *pSubresource* must only have a single bit set

- **VUID-vkGetImageSubresourceLayout-mipLevel-01716**
  
  The *mipLevel* member of *pSubresource* must be less than the *mipLevels* specified in *VkImageCreateInfo* when *image* was created

- **VUID-vkGetImageSubresourceLayout-arrayLayer-01717**
  
  The *arrayLayer* member of *pSubresource* must be less than the *arrayLayers* specified in *VkImageCreateInfo* when *image* was created

- **VUID-vkGetImageSubresourceLayout-format-04461**
  
  If *format* is a color format, the *aspectMask* member of *pSubresource* must be *VK_IMAGE_ASPECT_COLOR_BIT*

- **VUID-vkGetImageSubresourceLayout-format-04462**
  
  If *format* has a depth component, the *aspectMask* member of *pSubresource* must contain *VK_IMAGE_ASPECT_DEPTH_BIT*

- **VUID-vkGetImageSubresourceLayout-format-04463**
  
  If *format* has a stencil component, the *aspectMask* member of *pSubresource* must contain *VK_IMAGE_ASPECT_STENCIL_BIT*

- **VUID-vkGetImageSubresourceLayout-format-04464**
  
  If *format* does not contain a stencil or depth component, the *aspectMask* member of *pSubresource* must not contain *VK_IMAGE_ASPECT_DEPTH_BIT* or *VK_IMAGE_ASPECT_STENCIL_BIT*

- **VUID-vkGetImageSubresourceLayout-format-01581**
  
  If the *tiling* of the *image* is *VK_IMAGE_TILING_LINEAR* and its *format* is a multi-planar format with two planes, the *aspectMask* member of *pSubresource* must be *VK_IMAGE_ASPECT_PLANE_0_BIT* or *VK_IMAGE_ASPECT_PLANE_1_BIT*

- **VUID-vkGetImageSubresourceLayout-format-01582**
  
  If the *tiling* of the *image* is *VK_IMAGE_TILING_LINEAR* and its *format* is a multi-planar format with three planes, the *aspectMask* member of *pSubresource* must be *VK_IMAGE_ASPECT_PLANE_0_BIT*, *VK_IMAGE_ASPECT_PLANE_1_BIT* or *VK_IMAGE_ASPECT_PLANE_2_BIT*

- **VUID-vkGetImageSubresourceLayout-image-01895**
  
  If *image* was created with the *VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID* external memory handle type, then *image* must be bound to memory

- **VUID-vkGetImageSubresourceLayout-tiling-02271**
  
  If the *tiling* of the *image* is *VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT*, then the *aspectMask* member of *pSubresource* must be *VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT* and the index *i* must be less than the *VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount* associated with the image's *format* and *VkImageDrmFormatModifierPropertiesEXT::drmFormatModifier*.
Valid Usage (Implicit)

- VUID-vkGetImageSubresourceLayout-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkGetImageSubresourceLayout-image-parameter
  
  **image** must be a valid **VkImage** handle

- VUID-vkGetImageSubresourceLayout-pSubresource-parameter
  
  **pSubresource** must be a valid pointer to a valid **VkImageSubresource** structure

- VUID-vkGetImageSubresourceLayout-pLayout-parameter
  
  **pLayout** must be a valid pointer to a **VkSubresourceLayout** structure

- VUID-vkGetImageSubresourceLayout-image-parent
  
  **image** must have been created, allocated, or retrieved from **device**

The **VkImageSubresource** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageSubresource {
    VkImageAspectFlags aspectMask;
    uint32_t mipLevel;
    uint32_t arrayLayer;
} VkImageSubresource;
```

- **aspectMask** is a **VkImageAspectFlags** selecting the image *aspect*.
- **mipLevel** selects the mipmap level.
- **arrayLayer** selects the array layer.

Valid Usage (Implicit)

- VUID-VkImageSubresource-aspectMask-parameter
  
  **aspectMask** must be a valid combination of **VkImageAspectFlagBits** values

- VUID-VkImageSubresource-aspectMask-requiredbitmask
  
  **aspectMask** must not be 0

Information about the layout of the image subresource is returned in a **VkSubresourceLayout** structure:
// Provided by VK_VERSION_1_0

typedef struct VkSubresourceLayout {
    VkDeviceSize    offset;
    VkDeviceSize    size;
    VkDeviceSize    rowPitch;
    VkDeviceSize    arrayPitch;
    VkDeviceSize    depthPitch;
} VkSubresourceLayout;

- **offset** is the byte offset from the start of the image or the plane where the image subresource begins.
- **size** is the size in bytes of the image subresource. **size** includes any extra memory that is required based on **rowPitch**.
- **rowPitch** describes the number of bytes between each row of texels in an image.
- **arrayPitch** describes the number of bytes between each array layer of an image.
- **depthPitch** describes the number of bytes between each slice of 3D image.

If the image is **linear**, then **rowPitch**, **arrayPitch** and **depthPitch** describe the layout of the image subresource in linear memory. For uncompressed formats, **rowPitch** is the number of bytes between texels with the same x coordinate in adjacent rows (y coordinates differ by one). **arrayPitch** is the number of bytes between texels with the same x and y coordinate in adjacent array layers of the image (array layer values differ by one). **depthPitch** is the number of bytes between texels with the same x and y coordinate in adjacent slices of a 3D image (z coordinates differ by one). Expressed as an addressing formula, the starting byte of a texel in the image subresource has address:

```
// (x,y,z,layer) are in texel coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x*elementSize + offset
```

For compressed formats, the **rowPitch** is the number of bytes between compressed texel blocks in adjacent rows. **arrayPitch** is the number of bytes between compressed texel blocks in adjacent array layers. **depthPitch** is the number of bytes between compressed texel blocks in adjacent slices of a 3D image.

```
// (x,y,z,layer) are in compressed texel block coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x *compressedTexelBlockByteSize + offset;
```

The value of **arrayPitch** is undefined for images that were not created as arrays. **depthPitch** is defined only for 3D images.

If the image has a **single-plane** color format and its tiling is **VK_IMAGE_TILING_LINEAR**, then the aspectMask member of VkImageSubresource must be **VK_IMAGE_ASPECT_COLOR_BIT**.

If the image has a depth/stencil format and its tiling is **VK_IMAGE_TILING_LINEAR**, then aspectMask
**must** be either `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`. On implementations that store depth and stencil aspects separately, querying each of these image subresource layouts will return a different offset and size representing the region of memory used for that aspect. On implementations that store depth and stencil aspects interleaved, the same offset and size are returned and represent the interleaved memory allocation.

If the image has a **multi-planar format** and its tiling is `VK_IMAGE_TILING_LINEAR`, then the `aspectMask` member of `VkImageSubresource` **must** be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or (for 3-plane formats only) `VK_IMAGE_ASPECT_PLANE_2_BIT`. Querying each of these image subresource layouts will return a different offset and size representing the region of memory used for that plane. If the image is **disjoint**, then the offset is relative to the base address of the plane. If the image is **non-disjoint**, then the offset is relative to the base address of the image.

If the image’s tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the `aspectMask` member of `VkImageSubresource` **must** be one of `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT`, where the maximum allowed plane index i is defined by the `VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount` associated with the image’s `VkImageCreateInfo::format` and `modifier`. The memory range used by the subresource is described by offset and size. If the image is **disjoint**, then the offset is relative to the base address of the **memory plane**. If the image is **non-disjoint**, then the offset is relative to the base address of the image. If the image is **non-linear**, then `rowPitch`, `arrayPitch`, and `depthPitch` have an implementation-dependent meaning.

If an image was created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then the image has a **Linux DRM format modifier**. To query the **modifier**, call:

```c
// Provided by VK_EXT_image_drm_format_modifier
VkResult vkGetImageDrmFormatModifierPropertiesEXT(
    VkDevice device,                     // The logical device that owns the image.
    VkImage image,                      // The queried image.
    VkImageDrmFormatModifierPropertiesEXT* pProperties
    // Will return properties of the image's DRM format modifier.
);
```

**Valid Usage**

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-image-02272
  - `image` **must** have been created with **tiling** equal to `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`
Valid Usage (Implicit)

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-image-parameter
image must be a valid VkImage handle

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-pProperties-parameter
pProperties must be a valid pointer to a VkImageDrmFormatModifierPropertiesEXT structure

- VUID-vkGetImageDrmFormatModifierPropertiesEXT-image-parent
image must have been created, allocated, or retrieved from device

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY

The VkImageDrmFormatModifierPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkImageDrmFormatModifierPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint64_t drmFormatModifier;
} VkImageDrmFormatModifierPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `drmFormatModifier` returns the image's Linux DRM format modifier.

If the image was created with VkImageDrmFormatModifierListCreateInfoEXT, then the returned `drmFormatModifier` must belong to the list of modifiers provided at time of image creation in VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers. If the image was created with VkImageDrmFormatModifierExplicitCreateInfoEXT, then the returned `drmFormatModifier` must be the modifier provided at time of image creation in VkImageDrmFormatModifierExplicitCreateInfoEXT::drmFormatModifier.
Valid Usage (Implicit)

- **VUID-VkImageDrmFormatModifierPropertiesEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT`

- **VUID-VkImageDrmFormatModifierPropertiesEXT-pNext-pNext**
  
  `pNext` must be NULL

To destroy an image, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyImage(
    VkDevice device,
    VkImage image,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the image.
- `image` is the image to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- **VUID-vkDestroyImage-image-01000**
  
  All submitted commands that refer to `image`, either directly or via a `VkImageView`, must have completed execution

- **VUID-vkDestroyImage-image-01001**
  
  If `VkAllocationCallbacks` were provided when `image` was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyImage-image-01002**
  
  If no `VkAllocationCallbacks` were provided when `image` was created, `pAllocator` must be NULL

- **VUID-vkDestroyImage-image-04882**
  
  `image` must not have been acquired from `vkGetSwapchainImagesKHR`
Valid Usage (Implicit)

- VUID-vkDestroyImage-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkDestroyImage-image-parameter
  
  If **image** is not **VK_NULL_HANDLE**, **image** must be a valid **VkImage** handle

- VUID-vkDestroyImage-pAllocator-parameter
  
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkDestroyImage-image-parent
  
  If **image** is a valid handle, it must have been created, allocated, or retrieved from **device**

Host Synchronization

- Host access to **image** must be externally synchronized

12.3.1. Image Format Features

Valid uses of a **VkImage** may depend on the image’s **format features**, defined below. Such constraints are documented in the affected valid usage statement.

- If the image was created with **VK_IMAGE_TILING_LINEAR**, then its set of **format features** is the value of **VkFormatProperties::linearTilingFeatures** found by calling **vkGetPhysicalDeviceFormatProperties** on the same **format** as **VkImageCreateInfo::format**.

- If the image was created with **VK_IMAGE_TILING_OPTIMAL**, but without an Android hardware buffer external format, then its set of **format features** is the value of **VkFormatProperties::optimalTilingFeatures** found by calling **vkGetPhysicalDeviceFormatProperties** on the same **format** as **VkImageCreateInfo::format**.

- If the image was created with an Android hardware buffer external format, then its set of **format features** is the value of **VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures** found by calling **vkGetAndroidHardwareBufferPropertiesANDROID** on the Android hardware buffer that was imported to the **VkDeviceMemory** to which the image is bound.

- If the image was created with **VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT**, then:
  - The image’s DRM format modifier is the value of **VkImageDrmFormatModifierListCreateInfoEXT::drmFormatModifier** found by calling **vkGetImageDrmFormatModifierPropertiesEXT**.
  - Let **VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties** be the array found by calling **vkGetPhysicalDeviceFormatProperties2** on the same **format** as **VkImageCreateInfo::format**.
  - Let **VkDrmFormatModifierPropertiesEXT prop** be an array element whose **drmFormatModifier** member is the value of the image’s DRM format modifier.
Then the image set of format features is the value of taking the bitwise intersection over the collected \texttt{prop:drmFormatModifierTilingFeatures}.

### 12.3.2. Corner-Sampled Images

A \textit{corner-sampled image} is an image where unnormalized texel coordinates are centered on integer values rather than half-integer values.

A corner-sampled image has a number of differences compared to conventional texture image:

- Texels are centered on integer coordinates. See \textit{Unnormalized Texel Coordinate Operations}.
- Normalized coordinates are scaled using $\text{coord} \times (\text{dim} - 1)$ rather than $\text{coord} \times \text{dim}$, where \text{dim} is the size of one dimension of the image. See \textit{normalized texel coordinate transform}.
- Partial derivatives are scaled using $\text{coord} \times (\text{dim} - 1)$ rather than $\text{coord} \times \text{dim}$. See \textit{Scale Factor Operation}.
- Calculation of the next higher lod size goes according to $\lceil \text{dim} / 2 \rceil$ rather than $\lfloor \text{dim} / 2 \rfloor$. See \textit{Image Miplevel Sizing}.
- The minimum level size is 2x2 for 2D images and 2x2x2 for 3D images. See \textit{Image Miplevel Sizing}.

Corner-sampling is only supported for 2D and 3D images. When sampling a corner-sampled image, the sampler addressing mode \textbf{must} be \texttt{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE}. Corner-sampled images are not supported as cubemaps or depth/stencil images.

### 12.3.3. Image Miplevel Sizing

A \textit{complete mipmap chain} is the full set of miplevels, from the largest miplevel provided, down to the \textit{minimum mipmap size}.

**Conventional Images**

For conventional images, the dimensions of each successive miplevel, $n+1$, are:

\[
\text{width}_{n+1} = \max(\lfloor \text{width}_n / 2 \rfloor, 1)
\]

\[
\text{height}_{n+1} = \max(\lfloor \text{height}_n / 2 \rfloor, 1)
\]

\[
\text{depth}_{n+1} = \max(\lfloor \text{depth}_n / 2 \rfloor, 1)
\]

where \text{width}_n, \text{height}_n, and \text{depth}_n are the dimensions of the next larger miplevel, $n$.

The minimum mipmap size is:

- 1 for one-dimensional images,
- 1x1 for two-dimensional images, and
• 1x1x1 for three-dimensional images.

The number of levels in a complete mipmap chain is:

$$\lfloor \log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0)) \rfloor + 1$$

where \(\text{width}_0\), \(\text{height}_0\), and \(\text{depth}_0\) are the dimensions of the largest (most detailed) mipmap level, 0.

### Corner-Sampled Images

For corner-sampled images, the dimensions of each successive mipmap level, \(n+1\), are:

$$\text{width}_{n+1} = \max(\lceil \text{width}_n/2 \rceil, 2)$$

$$\text{height}_{n+1} = \max(\lceil \text{height}_n/2 \rceil, 2)$$

$$\text{depth}_{n+1} = \max(\lceil \text{depth}_n/2 \rceil, 2)$$

where \(\text{width}_n\), \(\text{height}_n\), and \(\text{depth}_n\) are the dimensions of the next larger mipmap level, \(n\).

The minimum mipmap level size is:

• 2x2 for two-dimensional images, and
• 2x2x2 for three-dimensional images.

The number of levels in a complete mipmap chain is:

$$\lceil \log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0)) \rceil$$

where \(\text{width}_0\), \(\text{height}_0\), and \(\text{depth}_0\) are the dimensions of the largest (most detailed) mipmap level, 0.

### 12.4. Image Layouts

Images are stored in implementation-dependent opaque layouts in memory. Each layout has limitations on what kinds of operations are supported for image subresources using the layout. At any given time, the data representing an image subresource in memory exists in a particular layout which is determined by the most recent layout transition that was performed on that image subresource. Applications have control over which layout each image subresource uses, and can transition an image subresource from one layout to another. Transitions can happen with an image memory barrier, included as part of a \(\text{vkCmdPipelineBarrier}\) or a \(\text{vkCmdWaitEvents}\) command buffer command (see Image Memory Barriers), or as part of a subpass dependency within a render pass (see \(\text{VkSubpassDependency}\)). The image layout is per-image subresource, and separate image subresources of the same image can be in different layouts at the same time with one exception - depth and stencil aspects of a given image subresource must always be in the same layout.
Note

Each layout may offer optimal performance for a specific usage of image memory. For example, an image with a layout of VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL may provide optimal performance for use as a color attachment, but be unsupported for use in transfer commands. Applications can transition an image subresource from one layout to another in order to achieve optimal performance when the image subresource is used for multiple kinds of operations. After initialization, applications need not use any layout other than the general layout, though this may produce suboptimal performance on some implementations.

Upon creation, all image subresources of an image are initially in the same layout, where that layout is selected by the VkImageCreateInfo::initialLayout member. The initialLayout must be either VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED. If it is VK_IMAGE_LAYOUT_PREINITIALIZED, then the image data can be preinitialized by the host while using this layout, and the transition away from this layout will preserve that data. If it is VK_IMAGE_LAYOUT_UNDEFINED, then the contents of the data are considered to be undefined, and the transition away from this layout is not guaranteed to preserve that data. For either of these initial layouts, any image subresources must be transitioned to another layout before they are accessed by the device.

Host access to image memory is only well-defined for linear images and for image subresources of those images which are currently in either the VK_IMAGE_LAYOUT_PREINITIALIZED or VK_IMAGE_LAYOUT_GENERAL layout. Calling vkGetImageSubresourceLayout for a linear image returns a subresource layout mapping that is valid for either of those image layouts.

The set of image layouts consists of:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageLayout {
    VK_IMAGE_LAYOUT_UNDEFINED = 0,
    VK_IMAGE_LAYOUT_GENERAL = 1,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL = 3,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
    VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
    VK_IMAGE_LAYOUT_PREINITIALIZED = 8,

    // Provided by VK_VERSION_1_1
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL = 1000117000,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL = 1000117001,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_ATTACHMENT_OPTIMAL = 1000241000,
    VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL = 1000241001,
    VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL = 1000241002,
    VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL = 1000241003,

    // Provided by VK_KHR_swapchain
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR = 1000001002,

    #ifdef VK_ENABLE_BETA_EXTENSIONS
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR = 1000001002,
    #endif
};
```
# Provided by VK_KHR_video_decode_queue

VK_IMAGE_LAYOUT_VIDEO_DECODE_DST_KHR = 1000024000,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
# Provided by VK_KHR_video_decode_queue

VK_IMAGE_LAYOUT_VIDEO_DECODE_SRC_KHR = 1000024001,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
# Provided by VK_KHR_video_decode_queue

VK_IMAGE_LAYOUT_VIDEO_DECODE_DPB_KHR = 1000024002,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
# Provided by VK_KHR_video_decode_queue

VK_IMAGE_LAYOUT_VIDEO_ENCODE_DST_KHR = 1000299000,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
# Provided by VK_KHR_video_encode_queue

VK_IMAGE_LAYOUT_VIDEO_ENCODE_SRC_KHR = 1000299001,
#endif

#ifdef VK_ENABLE_BETA_EXTENSIONS
# Provided by VK_KHR_video_encode_queue

VK_IMAGE_LAYOUT_VIDEO_ENCODE_DPB_KHR = 1000299002,
#endif

// Provided by VK_KHR_shared_presentable_image

VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR = 1000111000,

// Provided by VK_KHR_shared_presentable_image

VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV = 1000164003,

// Provided by VK_KHR_shading_rate_image

VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT = 1000218000,

// Provided by VK_KHR_synchronization2

VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR = 1000314000,

// Provided by VK_KHR_synchronization2

VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR = 1000314001,

// Provided by VK_KHR_maintenance2

VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL,

// Provided by VK_KHR_maintenance2

VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,

// Provided by VK_KHR_fragment_shading_rate

VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV,

// Provided by VK_KHR_separate_depth_stencil_layouts

VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,

// Provided by VK_KHR_separate_depth_stencil_layouts

VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL_KHR =
VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL,
The type(s) of device access supported by each layout are:

- **VK_IMAGE_LAYOUT_UNDEFINED** specifies that the layout is unknown. Image memory cannot be transitioned into this layout. This layout can be used as the initialLayout member of VkImageCreateInfo. This layout can be used in place of the current image layout in a layout transition, but doing so will cause the contents of the image's memory to be undefined.

- **VK_IMAGE_LAYOUT_PREINITIALIZED** specifies that an image's memory is in a defined layout and cannot be populated by data, but that it has not yet been initialized by the driver. Image memory cannot be transitioned into this layout. This layout can be used as the initialLayout member of VkImageCreateInfo. This layout is intended to be used as the initial layout for an image whose contents are written by the host, and hence the data can be written to memory immediately, without first executing a layout transition. Currently, VK_IMAGE_LAYOUT_PREINITIALIZED is only useful with linear images because there is not a standard layout defined for VK_IMAGE_TILING_OPTIMAL images.

- **VK_IMAGE_LAYOUT_GENERAL** supports all types of device access.

- **VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR** specifies a layout that must only be used with attachment accesses in the graphics pipeline.

- **VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR** specifies a layout allowing read only access as an attachment, or in shaders as a sampled image, combined image/sampler, or input attachment.

- **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** must only be used as a color or resolve attachment in a VkFramebuffer. This layout is valid only for image subresources of images created with the VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT usage bit enabled.

- **VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL** specifies a layout for both the depth and stencil aspects of a depth/stencil format image allowing read and write access as a depth/stencil attachment. It is equivalent to VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL and VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL.

- **VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL** specifies a layout for both the depth and stencil aspects of a depth/stencil format image allowing read only access as a depth/stencil attachment or in shaders as a sampled image, combined image/sampler, or input attachment. It is equivalent to VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL and VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL.

- **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL** specifies a layout for depth/stencil format images allowing read and write access to the stencil aspect as a stencil attachment, and read only access to the depth aspect as a depth attachment or in shaders as a sampled image, combined image/sampler, or input attachment. It is equivalent to VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL and VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL.

- **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL** specifies a layout for depth/stencil format images allowing read and write access to the stencil aspect as a stencil attachment, and read only access to the depth aspect as a depth attachment, and read only access to the stencil aspect as a stencil attachment or in shaders as a sampled image,
combined image/sampler, or input attachment. It is equivalent to
\texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_ATTACHMENT\_OPTIMAL} and \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_READ\_ONLY\_OPTIMAL}.

- \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_ATTACHMENT\_OPTIMAL} specifies a layout for the depth aspect of a
  depth/stencil format image allowing read and write access as a depth attachment.
- \texttt{VK\_IMAGE\_LAYOUT\_DEPTH\_READ\_ONLY\_OPTIMAL} specifies a layout for the depth aspect of a
  depth/stencil format image allowing read-only access as a depth attachment or in shaders as a
  sampled image, combined image/sampler, or input attachment.
- \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_ATTACHMENT\_OPTIMAL} specifies a layout for the stencil aspect of a
  depth/stencil format image allowing read and write access as a stencil attachment.
- \texttt{VK\_IMAGE\_LAYOUT\_STENCIL\_READ\_ONLY\_OPTIMAL} specifies a layout for the stencil aspect of a
  depth/stencil format image allowing read-only access as a stencil attachment or in shaders as a
  sampled image, combined image/sampler, or input attachment.
- \texttt{VK\_IMAGE\_LAYOUT\_SHADER\_READ\_ONLY\_OPTIMAL} specifies a layout allowing read-only access in a
  shader as a sampled image, combined image/sampler, or input attachment. This layout is valid
  only for image subresources of images created with the \texttt{VK\_IMAGE\_USAGE\_SAMPLED\_BIT} or
  \texttt{VK\_IMAGE\_USAGE\_INPUT\_ATTACHMENT\_BIT} usage bit enabled.
- \texttt{VK\_IMAGE\_LAYOUT\_TRANSFER\_SRC\_OPTIMAL} must only be used as a source image of a transfer
  command (see the definition of \texttt{VK\_PIPELINE\_STAGE\_TRANSFER\_BIT}). This layout is valid only for
  image subresources of images created with the \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_SRC\_BIT} usage bit
  enabled.
- \texttt{VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL} must only be used as a destination image of a transfer
  command. This layout is valid only for image subresources of images created with the
  \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT} usage bit enabled.
- \texttt{VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR} must only be used for presenting a presentable image for
display. A swapchain's image must be transitioned to this layout before calling
\texttt{vkQueuePresentKHR}, and must be transitioned away from this layout after calling
\texttt{vkAcquireNextImageKHR}.
- \texttt{VK\_IMAGE\_LAYOUT\_SHARED\_PRESENT\_KHR} is valid only for shared presentable images, and must be
  used for any usage the image supports.
- \texttt{VK\_IMAGE\_LAYOUT\_FRAGMENT\_SHADING\_RATE\_ATTACHMENT\_OPTIMAL\_KHR} must only be used as a
  fragment shading rate attachment or shading rate image. This layout is valid only for image
  subresources of images created with the \texttt{VK\_IMAGE\_USAGE\_FRAGMENT\_SHADING\_RATE\_ATTACHMENT\_BIT\_KHR} usage bit enabled.
- \texttt{VK\_IMAGE\_LAYOUT\_FRAGMENT\_DENSITY\_MAP\_OPTIMAL\_EXT} must only be used as a fragment density
  map attachment in a \texttt{VkRenderPass}. This layout is valid only for image subresources of images
  created with the \texttt{VK\_IMAGE\_USAGE\_FRAGMENT\_DENSITY\_MAP\_BIT\_EXT} usage bit enabled.
- \texttt{VK\_IMAGE\_LAYOUT\_VIDEO\_DECODE\_DST\_KHR} must only be used as a decode output image of a \texttt{video
  decode operation}. This layout is valid only for image subresources of images created with the
  \texttt{VK\_IMAGE\_USAGE\_VIDEO\_DECODE\_DST\_BIT\_KHR} usage bit enabled.
- \texttt{VK\_IMAGE\_LAYOUT\_VIDEO\_DECODE\_SRC\_KHR} must only be used as a decode source image of a \texttt{video
  decode operation}. This layout is valid only for image subresources of images created with the
  \texttt{VK\_IMAGE\_USAGE\_VIDEO\_DECODE\_SRC\_BIT\_KHR} usage bit enabled.
• **VK_IMAGE_LAYOUT_VIDEO_DECODE_DPB_KHR** must only be used as a decode source or destination image of a *video decode operation*. This layout is valid only for image subresources of images created with the **VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR** usage bit enabled.

• **VK_IMAGE_LAYOUT_VIDEO_ENCODE_DST_KHR** must only be used as a encode output image of a *video encode operation*. This layout is valid only for image subresources of images created with the **VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR** usage bit enabled.

• **VK_IMAGE_LAYOUT_VIDEO_ENCODE_SRC_KHR** must only be used as a encode source image of a *video encode operation*. This layout is valid only for image subresources of images created with the **VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR** usage bit enabled.

• **VK_IMAGE_LAYOUT_VIDEO_ENCODE_DPB_KHR** must only be used as a encode source or destination image of a *video encode operation*. This layout is valid only for image subresources of images created with the **VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR** usage bit enabled.

The layout of each image subresource is not a state of the image subresource itself, but is rather a property of how the data in memory is organized, and thus for each mechanism of accessing an image in the API the application must specify a parameter or structure member that indicates which image layout the image subresource(s) are considered to be in when the image will be accessed. For transfer commands, this is a parameter to the command (see Clear Commands and Copy Commands). For use as a framebuffer attachment, this is a member in the substructures of the **VkRenderPassCreateInfo** (see Render Pass). For use in a descriptor set, this is a member in the **VkDescriptorImageInfo** structure (see Descriptor Set Updates).

### 12.4.1. Image Layout Matching Rules

At the time that any command buffer command accessing an image executes on any queue, the layouts of the image subresources that are accessed must all match exactly the layout specified via the API controlling those accesses, except in case of accesses to an image with a depth/stencil format performed through descriptors referring to only a single aspect of the image, where the following relaxed matching rules apply:

• Descriptors referring just to the depth aspect of a depth/stencil image only need to match in the image layout of the depth aspect, thus **VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL** and **VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL** are considered to match.

• Descriptors referring just to the stencil aspect of a depth/stencil image only need to match in the image layout of the stencil aspect, thus **VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL** and **VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL** are considered to match.

When performing a layout transition on an image subresource, the old layout value must either equal the current layout of the image subresource (at the time the transition executes), or else **VK_IMAGE_LAYOUT_UNDEFINED** (implying that the contents of the image subresource need not be preserved). The new layout used in a transition must not be **VK_IMAGE_LAYOUT_UNDEFINED** or **VK_IMAGE_LAYOUT_PREINITIALIZED**.

The image layout of each image subresource of a depth/stencil image created with **VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT** is dependent on the last sample locations used to render to the image subresource as a depth/stencil attachment, thus applications must provide the same sample locations that were last used to render to the given image
subresource whenever a layout transition of the image subresource happens, otherwise the contents of the depth aspect of the image subresource become undefined.

In addition, depth reads from a depth/stencil attachment referring to an image subresource range of a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` using different sample locations than what have been last used to perform depth writes to the image subresources of the same image subresource range return undefined values.

Similarly, depth writes to a depth/stencil attachment referring to an image subresource range of a depth/stencil image created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` using different sample locations than what have been last used to perform depth writes to the image subresources of the same image subresource range make the contents of the depth aspect of those image subresources undefined.

12.5. Image Views

Image objects are not directly accessed by pipeline shaders for reading or writing image data. Instead, *image views* representing contiguous ranges of the image subresources and containing additional metadata are used for that purpose. Views must be created on images of compatible types, and must represent a valid subset of image subresources.

Image views are represented by `VkImageView` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImageView)
```

The types of image views that can be created are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageViewType {
    VK_IMAGE_VIEW_TYPE_1D = 0,
    VK_IMAGE_VIEW_TYPE_2D = 1,
    VK_IMAGE_VIEW_TYPE_3D = 2,
    VK_IMAGE_VIEW_TYPE_CUBE = 3,
    VK_IMAGE_VIEW_TYPE_1D_ARRAY = 4,
    VK_IMAGE_VIEW_TYPE_2D_ARRAY = 5,
    VK_IMAGE_VIEW_TYPE_CUBE_ARRAY = 6,
} VkImageViewType;
```

The exact image view type is partially implicit, based on the image's type and sample count, as well as the view creation parameters as described in the image view compatibility table for `vkCreateImageView`. This table also shows which SPIR-V `OpTypeImage Dim` and `Arrayed` parameters correspond to each image view type.

To create an image view, call:
// Provided by VK_VERSION_1_0
VkResult vkCreateImageView(
    VkDevice                                    device,
    const VkImageViewCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImageView*                                pView);

- **device** is the logical device that creates the image view.
- **pCreateInfo** is a pointer to a `VkImageViewCreateInfo` structure containing parameters to be used to create the image view.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pView** is a pointer to a `VkImageView` handle in which the resulting image view object is returned.

### Valid Usage (Implicit)

- VUID-vkCreateImageView-device-parameter  
  **device** must be a valid `VkDevice` handle
- VUID-vkCreateImageView-pCreateInfo-parameter  
  **pCreateInfo** must be a valid pointer to a valid `VkImageViewCreateInfo` structure
- VUID-vkCreateImageView-pAllocator-parameter  
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure
- VUID-vkCreateImageView-pView-parameter  
  **pView** must be a valid pointer to a `VkImageView` handle

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkImageViewCreateInfo` structure is defined as:
// Provided by VK_VERSION_1_0
typedef struct VkImageViewCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkImageViewCreateFlags flags;
    VkImage image;
    VkImageViewType viewType;
    VkFormat format;
    VkComponentMapping components;
    VkImageSubresourceRange subresourceRange;
} VkImageViewCreateInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is a bitmask of VkImageViewCreateFlagBits describing additional parameters of the image view.
• image is a VkImage on which the view will be created.
• viewType is a VkImageViewType value specifying the type of the image view.
• format is a VkFormat describing the format and type used to interpret texel blocks in the image.
• components is a VkComponentMapping specifying a remapping of color components (or of depth or stencil components after they have been converted into color components).
• subresourceRange is a VkImageSubresourceRange selecting the set of mipmap levels and array layers to be accessible to the view.

Some of the image creation parameters are inherited by the view. In particular, image view creation inherits the implicit parameter usage specifying the allowed usages of the image view that, by default, takes the value of the corresponding usage parameter specified in VkImageCreateInfo at image creation time. The implicit usage can be overridden by adding a VkImageViewUsageCreateInfo structure to the pNext chain, but the view usage must be a subset of the image usage. If the image was has a depth-stencil format and was created with a VkImageStencilUsageCreateInfo structure included in the pNext chain of VkImageCreateInfo, the usage is calculated based on the subresource.aspectMask provided:

• If aspectMask includes only VK_IMAGE_ASPECT_STENCIL_BIT, the implicit usage is equal to VkImageStencilUsageCreateInfo::stencilUsage.
• If aspectMask includes only VK_IMAGE_ASPECT_DEPTH_BIT, the implicit usage is equal to VkImageCreateInfo::usage.
• If both aspects are included in aspectMask, the implicit usage is equal to the intersection of VkImageCreateInfo::usage and VkImageStencilUsageCreateInfo::stencilUsage.

If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, and if the format of the image is not multi-planar, format can be different from the image's format, but if image was created without the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag and they are not equal they must be compatible. Image format compatibility is defined in the Format Compatibility Classes section.
Views of compatible formats will have the same mapping between texel coordinates and memory locations irrespective of the format, with only the interpretation of the bit pattern changing.

**Note**

Values intended to be used with one view format may not be exactly preserved when written or read through a different format. For example, an integer value that happens to have the bit pattern of a floating point denorm or NaN may be flushed or canonicalized when written or read through a view with a floating point format. Similarly, a value written through a signed normalized format that has a bit pattern exactly equal to \( -2^b \) may be changed to \( -2^b + 1 \) as described in Conversion from Normalized Fixed-Point to Floating-Point.

If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag, format must be compatible with the image’s format as described above, or must be an uncompressed format in which case it must be size-compatible with the image’s format, as defined for copying data between images. In this case the resulting image view’s texel dimensions equal the dimensions of the selected mip level divided by the compressed texel block size and rounded up.

The VkComponentMapping components member describes a remapping from components of the image to components of the vector returned by shader image instructions. This remapping must be the identity swizzle for storage image descriptors, input attachment descriptors, framebuffer attachments, and any VkImageView used with a combined image sampler that enables sampler Y’C\(b\)

If the image view is to be used with a sampler which supports sampler Y’C\(b\) conversion, an identically defined object of type VkSamplerYcbcrConversion to that used to create the sampler must be passed to vkCreateImageView in a VkSamplerYcbcrConversionInfo included in the pNext chain of VkImageViewCreateInfo. Conversely, if a VkSamplerYcbcrConversion object is passed to vkCreateImageView, an identically defined VkSamplerYcbcrConversion object must be used when sampling the image.

If the image has a multi-planar format and subresourceRange.aspectMask is VK_IMAGE_ASPECT_COLOR_BIT, and image has been created with a usage value not containing any of the VK_IMAGE_USAGE_VIDEO_DECODER_DST_BIT_KHR, VK_IMAGE_USAGE_VIDEO_DECODER_SRC_BIT_KHR, VK_IMAGE_USAGE_VIDEO_DECODER_DBP_BIT_KHR, VK_IMAGE_USAGE_VIDEO_ENCODER_DST_BIT_KHR, and VK_IMAGE_USAGE_VIDEO_ENCODER_DBP_BIT_KHR flags, then the format must be identical to the image format, and the sampler to be used with the image view must enable sampler Y’C\(b\) conversion.

If the image has a multi-planar format and the image has been created with a usage value containing any of the VK_IMAGE_USAGE_VIDEO_DECODER_DST_BIT_KHR, VK_IMAGE_USAGE_VIDEO_DECODER_SRC_BIT_KHR, and VK_IMAGE_USAGE_VIDEO_DECODER_DBP_BIT_KHR flags, then all of the video decode operations would ignore the VkSamplerYcbcrConversionInfo structure and/or sampler Y’C\(b\) conversion object, associated with the image view. If the image has a multi-planar format and the image has been created with a usage value containing any of the VK_IMAGE_USAGE_VIDEO_ENCODER_DST_BIT_KHR, VK_IMAGE_USAGE_VIDEO_ENCODER_SRC_BIT_KHR, and VK_IMAGE_USAGE_VIDEO_ENCODER_DBP_BIT_KHR flags, then all of the video encode operations would ignore the VkSamplerYcbcrConversionInfo structure and/or sampler Y’C\(b\) conversion object, associated with the image view.
If `image` was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` and the image has a multi-planar format, and if `subresourceRange.aspectMask` is `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`, format must be compatible with the corresponding plane of the image, and the sampler to be used with the image view must not enable sampler Y'CbCr conversion. The width and height of the single-plane image view must be derived from the multi-planar image’s dimensions in the manner listed for plane compatibility for the plane.

Any view of an image plane will have the same mapping between texel coordinates and memory locations as used by the channels of the color aspect, subject to the formulae relating texel coordinates to lower-resolution planes as described in Chroma Reconstruction. That is, if an R or B plane has a reduced resolution relative to the G plane of the multi-planar image, the image view operates using the \((u_{\text{plane}}, v_{\text{plane}})\) unnormalized coordinates of the reduced-resolution plane, and these coordinates access the same memory locations as the \((u_{\text{color}}, v_{\text{color}})\) unnormalized coordinates of the color aspect for which chroma reconstruction operations operate on the same \((i_{\text{plane}}, j_{\text{plane}})\) coordinates.

Table 16. Image and image view parameter compatibility requirements

<table>
<thead>
<tr>
<th>Dim, Arrayed, MS</th>
<th>Image parameters</th>
<th>View parameters</th>
</tr>
</thead>
</table>
| **imageType** = `ci.imageType`  
width = `ci.extent.width`  
height = `ci.extent.height`  
depth = `ci.extent.depth`  
arrayLayers = `ci.arrayLayers`  
samples = `ci.samples`  
flags = `ci.flags`  
where `ci` is the `VkImageCreateInfo` used to create `image`. | baseArrayLayer, layerCount, and levelCount are members of the `subresourceRange` member. | |
| **1D, 0, 0**  
`imageType` = `VK_IMAGE_TYPE_1D`  
width ≥ 1  
height = 1  
depth = 1  
arrayLayers ≥ 1  
samples = 1 | viewType = `VK_IMAGE_VIEW_TYPE_1D`  
baseArrayLayer ≥ 0  
layerCount = 1 | |
| **1D, 1, 0**  
`imageType` = `VK_IMAGE_TYPE_1D`  
width ≥ 1  
height = 1  
depth = 1  
arrayLayers ≥ 1  
samples = 1 | viewType = `VK_IMAGE_VIEW_TYPE_1D_ARRAY`  
baseArrayLayer ≥ 0  
layerCount ≥ 1 | |
<table>
<thead>
<tr>
<th>Dim, Arrayed, MS</th>
<th>Image parameters</th>
<th>View parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D, 0, 0</td>
<td>图像类型 = VK_IMAGE_TYPE_2D</td>
<td>viewType = VK_IMAGE_VIEW_TYPE_2D</td>
</tr>
<tr>
<td></td>
<td>width ≥ 1</td>
<td>baseArrayLayer ≥ 0</td>
</tr>
<tr>
<td></td>
<td>height ≥ 1</td>
<td>layerCount = 1</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrayLayers ≥ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples = 1</td>
<td></td>
</tr>
<tr>
<td>2D, 1, 0</td>
<td>图像类型 = VK_IMAGE_TYPE_2D</td>
<td>viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY</td>
</tr>
<tr>
<td></td>
<td>width ≥ 1</td>
<td>baseArrayLayer ≥ 0</td>
</tr>
<tr>
<td></td>
<td>height ≥ 1</td>
<td>layerCount ≥ 1</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrayLayers ≥ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples = 1</td>
<td></td>
</tr>
<tr>
<td>2D, 0, 1</td>
<td>图像类型 = VK_IMAGE_TYPE_2D</td>
<td>viewType = VK_IMAGE_VIEW_TYPE_2D</td>
</tr>
<tr>
<td></td>
<td>width ≥ 1</td>
<td>baseArrayLayer ≥ 0</td>
</tr>
<tr>
<td></td>
<td>height ≥ 1</td>
<td>layerCount = 1</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrayLayers ≥ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples = 1</td>
<td></td>
</tr>
<tr>
<td>2D, 1, 1</td>
<td>图像类型 = VK_IMAGE_TYPE_2D</td>
<td>viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY</td>
</tr>
<tr>
<td></td>
<td>width ≥ 1</td>
<td>baseArrayLayer ≥ 0</td>
</tr>
<tr>
<td></td>
<td>height ≥ 1</td>
<td>layerCount ≥ 1</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrayLayers ≥ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples &gt; 1</td>
<td></td>
</tr>
<tr>
<td>CUBE, 0, 0</td>
<td>图像类型 = VK_IMAGE_TYPE_2D</td>
<td>viewType = VK_IMAGE_VIEW_TYPE_CUBE</td>
</tr>
<tr>
<td></td>
<td>width ≥ 1</td>
<td>baseArrayLayer ≥ 0</td>
</tr>
<tr>
<td></td>
<td>height = width</td>
<td>layerCount = 6</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrayLayers ≥ 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flags includes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT</td>
<td></td>
</tr>
<tr>
<td>CUBE, 1, 0</td>
<td>图像类型 = VK_IMAGE_TYPE_2D</td>
<td>viewType = VK_IMAGE_VIEW_TYPE_CUBE_ARRAY</td>
</tr>
<tr>
<td></td>
<td>width ≥ 1</td>
<td>baseArrayLayer ≥ 0</td>
</tr>
<tr>
<td></td>
<td>height = width</td>
<td>layerCount = 6 × N, N ≥ 1</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N ≥ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrayLayers ≥ 6 × N</td>
<td></td>
</tr>
<tr>
<td></td>
<td>samples = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flags includes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT</td>
<td></td>
</tr>
<tr>
<td>Dim, Arrayed, MS</td>
<td>Image parameters</td>
<td>View parameters</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
<td>-----------------</td>
</tr>
</tbody>
</table>
| 3D, 0, 0        | imageType = VK_IMAGE_TYPE_3D  
width ≥ 1  
height ≥ 1  
depth ≥ 1  
arrayLayers = 1  
samples = 1  
flags includes  
VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT  
flags does not include  
VK_IMAGE_CREATE_SPARSE_BINDING_BIT,  
VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and  
VK_IMAGE_CREATE_SPARSE_ALIASED_BIT | viewType = VK_IMAGE_VIEW_TYPE_3D  
baseArrayLayer = 0  
layerCount = 1 |
| 3D, 0, 0        | imageType = VK_IMAGE_TYPE_3D  
width ≥ 1  
height ≥ 1  
depth ≥ 1  
arrayLayers = 1  
samples = 1  
flags includes  
VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT  
flags does not include  
VK_IMAGE_CREATE_SPARSE_BINDING_BIT,  
VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and  
VK_IMAGE_CREATE_SPARSE_ALIASED_BIT | viewType = VK_IMAGE_VIEW_TYPE_2D  
levelCount = 1  
baseArrayLayer ≥ 0  
layerCount = 1 |
| 3D, 0, 0        | imageType = VK_IMAGE_TYPE_3D  
width ≥ 1  
height ≥ 1  
depth ≥ 1  
arrayLayers = 1  
samples = 1  
flags includes  
VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT  
flags does not include  
VK_IMAGE_CREATE_SPARSE_BINDING_BIT,  
VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, and  
VK_IMAGE_CREATE_SPARSE_ALIASED_BIT | viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY  
levelCount = 1  
baseArrayLayer ≥ 0  
layerCount ≥ 1 |
Valid Usage

- **VUID-VkImageViewCreateInfo-image-01003**
  
  If `image` was not created with `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` then `viewType` must not be `VK_IMAGE_VIEW_TYPE_CUBE` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`.

- **VUID-VkImageViewCreateInfo-viewType-01004**
  
  If the `image cubemap arrays` feature is not enabled, `viewType` must not be `VK_IMAGE_VIEW_TYPE_CUBE` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`.

- **VUID-VkImageViewCreateInfo-image-01005**
  
  If `image` was created with `VK_IMAGE_TYPE_3D` but without `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT` set then `viewType` must not be `VK_IMAGE_VIEW_TYPE_2D` or `VK_IMAGE_VIEW_TYPE_2D_ARRAY`.

- **VUID-VkImageViewCreateInfo-image-04441**
  
  `image` must have been created with a `usage` value containing at least one of the usages defined in the `valid image usage` list for image views.

- **VUID-VkImageViewCreateInfo-usage-02273**
  
  The `format features` of the resultant image view must contain at least one bit.

- **VUID-VkImageViewCreateInfo-usage-02274**
  
  If `usage` contains `VK_IMAGE_USAGE_SAMPLED_BIT`, then the `format features` of the resultant image view must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`.

- **VUID-VkImageViewCreateInfo-usage-02275**
  
  If `usage` contains `VK_IMAGE_USAGE_STORAGE_BIT`, then the image view's `format features` must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`.

- **VUID-VkImageViewCreateInfo-usage-02276**
  
  If `usage` contains `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT`, then the image view's `format features` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.

- **VUID-VkImageViewCreateInfo-usage-02277**
  
  If `usage` contains `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT`, then the image view's `format features` must contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- **VUID-VkImageViewCreateInfo-usage-02652**
  
  If `usage` contains `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT`, then the image view's `format features` must contain at least one of `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` or `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`.

- **VUID-VkImageViewCreateInfo-subresourceRange-01478**
  
  `subresourceRange.baseMipLevel` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

- **VUID-VkImageViewCreateInfo-subresourceRange-01718**
  
  If `subresourceRange.levelCount` is not `VK_REMAINING_MIP_LEVELS`, `subresourceRange.baseMipLevel + subresourceRange.levelCount` must be less than or equal to the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

- **VUID-VkImageViewCreateInfo-image-02571**
  
  If `image` was created with `usage` containing `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`, `subresourceRange.levelCount` must be 1.
If image is not a 3D image created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, or viewType is not VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY, subresourceRange.baseArrayLayer must be less than the arrayLayers specified in VkImageCreateInfo when image was created.

If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, image is not a 3D image created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, or viewType is not VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY, subresourceRange.layerCount must be non-zero and subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the arrayLayers specified in VkImageCreateInfo when image was created.

If image is a 3D image created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, and viewType is VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY, subresourceRange.baseArrayLayer must be less than the depth computed from baseMipLevel and extent.depth specified in VkImageCreateInfo when image was created, according to the formula defined in Image Miplevel Sizing.

If subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, image is a 3D image created with VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT set, and viewType is VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY, subresourceRange.layerCount must be non-zero and subresourceRange.baseArrayLayer + subresourceRange.layerCount must be less than or equal to the depth computed from baseMipLevel and extent.depth specified in VkImageCreateInfo when image was created, according to the formula defined in Image Miplevel Sizing.

If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, but without the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag, and if the format of the image is not a multi-planar format, format must be compatible with the format used to create image, as defined in Format Compatibility Classes.

If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag, format must be compatible with, or must be an uncompressed format that is size-compatible with, the format used to create image.

If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag, the levelCount and layerCount members of subresourceRange must both be 1.

If image was created with the VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT flag and format is a non-compressed format, viewType must not be VK_IMAGE_VIEW_TYPE_3D.

If a VkImageFormatListCreateInfo structure was included in the pNext chain of the VkImageCreateInfo structure used when creating image and VkImageFormatListCreateInfo::viewFormatCount is not zero then format must be one of the
formats in `VkImageFormatListCreateInfo::pViewFormats`

- **VUID-VkImageViewCreateInfo-image-01586**
  If *image* was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, if the *format* of the *image* is a multi-planar format, and if `subresourceRange.aspectMask` is one of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT, then *format* must be compatible with the *VkFormat* for the plane of the *image format* indicated by `subresourceRange.aspectMask`, as defined in Compatible formats of planes of multi-planar formats.

- **VUID-VkImageViewCreateInfo-image-01762**
  If *image* was not created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, or if the *format* of the *image* is a multi-planar format and if `subresourceRange.aspectMask` is VK_IMAGE_ASPECT_COLOR_BIT, *format* must be identical to the *format* used to create *image*.

- **VUID-VkImageViewCreateInfo-format-04724**
  If *format* is one of those listed in Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views, then the pNext chain must include a `VkSamplerYcbcrConversionInfo` structure with a conversion value other than VK_NULL_HANDLE.

- **VUID-VkImageViewCreateInfo-format-04714**
  If *format* has a _422_ or _420_ suffix then *image* must have been created with a width that is a multiple of 2.

- **VUID-VkImageViewCreateInfo-format-04715**
  If *format* has a _420_ suffix then *image* must have been created with a height that is a multiple of 2.

- **VUID-VkImageViewCreateInfo-pNext-01970**
  If the pNext chain includes a `VkSamplerYcbcrConversionInfo` structure with a conversion value other than VK_NULL_HANDLE, all members of components must have the identity swizzle.

- **VUID-VkImageViewCreateInfo-image-01020**
  If *image* is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-VkImageViewCreateInfo-subResourceRange-01021**
  `subresourceRange` and `viewType` must be compatible with the image, as described in the compatibility table.

- **VUID-VkImageViewCreateInfo-image-02399**
  If *image* has an external format, *format* must be VK_FORMAT_UNDEFINED.

- **VUID-VkImageViewCreateInfo-image-02400**
  If *image* has an external format, the pNext chain must include a `VkSamplerYcbcrConversionInfo` structure with a conversion object created with the same external format as *image*.

- **VUID-VkImageViewCreateInfo-image-02401**
  If *image* has an external format, all members of components must be the identity swizzle.

- **VUID-VkImageViewCreateInfo-image-02086**
  If *image* was created with usage containing...
If the shadingRateImage feature is enabled, and if image was created with usage containing VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV, format must be VK_FORMAT_R8_UINT

If the attachmentFragmentShadingRate feature is enabled, and the usage for the image view includes VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, then the image view's format features must contain VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR

If the attachmentFragmentShadingRate feature is enabled, the usage for the image view includes VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR, and layeredShadingRateAttachments is VK_FALSE, subresourceRange.layerCount must be 1

If dynamic fragment density map feature is not enabled, flags must not contain VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT

If deferred fragment density map feature is not enabled, flags must not contain VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DEFERRED_BIT_EXT

If flags contains VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DEFERRED_BIT_EXT, flags must not contain VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT

If image was created with flags containing VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT and usage containing VK_IMAGE_USAGE_SAMPLED_BIT, subresourceRange.layerCount must be less than or equal to VkPhysicalDeviceFragmentDensityMap2PropertiesEXT::maxSubsampledArrayLayers

If flags does not contain VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT and image was created with usage containing VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT, its flags must not contain any of VK_IMAGE_CREATE_PROTECTED_BIT, VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT

If the pNext chain includes a VkImageViewUsageCreateInfo structure, its usage member must not include any bits that were not set in the usage member of the VkImageCreateInfo structure used to create image

If the pNext chain includes a VkImageViewUsageCreateInfo structure, and image was not created with a VkImageStencilUsageCreateInfo structure included in the pNext chain of VkImageCreateInfo, its usage member must not include any bits that were not set in the usage member of the VkImageCreateInfo structure used to create image

If the pNext chain includes a VkImageViewUsageCreateInfo structure, image was created with a VkImageStencilUsageCreateInfo structure included in the pNext chain of
 VkImageCreateInfo, and subResourceRange.aspectMask includes VK_IMAGE_ASPECT_STENCIL_BIT, the usage member of the VkImageViewUsageCreateInfo instance must not include any bits that were not set in the usage member of the VkImageStenciUsageCreateInfo structure used to create image

- VUID-VkImageViewCreateInfo-pNext-02664
  If the pNext chain includes a VkImageViewUsageCreateInfo structure, image was created with a VkImageStenciUsageCreateInfo structure included in the pNext chain of VkImageCreateInfo, and subResourceRange.aspectMask includes bits other than VK_IMAGE_ASPECT_STENCIL_BIT, the usage member of the VkImageViewUsageCreateInfo structure must not include any bits that were not set in the usage member of the VkImageStenciUsageCreateInfo structure used to create image

- VUID-VkImageViewCreateInfo-viewType-02960
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE and subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.layerCount must be 6

- VUID-VkImageViewCreateInfo-viewType-02961
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE_ARRAY and subresourceRange.layerCount is not VK_REMAINING_ARRAY_LAYERS, subresourceRange.layerCount must be a multiple of 6

- VUID-VkImageViewCreateInfo-viewType-02962
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE and subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS, the remaining number of layers must be 6

- VUID-VkImageViewCreateInfo-viewType-02963
  If viewType is VK_IMAGE_VIEW_TYPE_CUBE_ARRAY and subresourceRange.layerCount is VK_REMAINING_ARRAY_LAYERS, the remaining number of layers must be a multiple of 6

- VUID-VkImageViewCreateInfo-imageViewFormatSwizzle-04465
  If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::imageViewFormatSwizzle is VK_FALSE, all elements of components must be VK_COMPONENT_SWIZZLE_IDENTITY.

- VUID-VkImageViewCreateInfo-imageViewFormatReinterpretation-04466
  If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::imageViewFormatReinterpretation is VK_FALSE, the VkFormat in format must not contain a different number of components, or a different number of bits in each component, than the format of the VkImage in image.

- VUID-VkImageViewCreateInfo-image-04817
  If image was created with usage containing VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR, VK_IMAGE_USAGE_VIDEO_DECODE_SRC_BIT_KHR, VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR, then the viewType must be VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY and all members of components must have the identity swizzle

- VUID-VkImageViewCreateInfo-image-04818
  If image was created with usage containing VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR, VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR, VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR, then the viewType must be VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY and all members of components must have the identity swizzle
Valid Usage (Implicit)

- **VUID-VkImageViewCreateInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO`

- **VUID-VkImageViewCreateInfo-pNext-pNext**
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either NULL or a pointer to a valid instance of `VkImageViewASTCDecodeModeEXT`, `VkImageViewUsageCreateInfo`, `VkSamplerYcbcrConversionInfo`, `VkVideoProfileKHR`, or `VkVideoProfilesKHR`

- **VUID-VkImageViewCreateInfo-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkImageViewCreateInfo-flags-parameter**
  - `flags` must be a valid combination of `VkImageViewCreateFlagBits` values

- **VUID-VkImageViewCreateInfo-image-parameter**
  - `image` must be a valid `VkImage` handle

- **VUID-VkImageViewCreateInfo-viewType-parameter**
  - `viewType` must be a valid `VkImageViewType` value

- **VUID-VkImageViewCreateInfo-format-parameter**
  - `format` must be a valid `VkFormat` value

- **VUID-VkImageViewCreateInfo-components-parameter**
  - `components` must be a valid `VkComponentMapping` structure

- **VUID-VkImageViewCreateInfo-subresourceRange-parameter**
  - `subresourceRange` must be a valid `VkImageSubresourceRange` structure

Bits which can be set in `VkImageViewCreateInfo::flags`, specifying additional parameters of an image, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkImageViewCreateFlagBits {
    // Provided by VK_EXT_fragment_density_map
    VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT = 0x00000001,
    // Provided by VK_EXT_fragment_density_map2
    VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DEFERRED_BIT_EXT = 0x00000002,
} VkImageViewCreateFlagBits;
```

- **VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT** specifies that the fragment density map will be read by device during `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`
- **VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DEFERRED_BIT_EXT** specifies that the fragment density map will be read by the host during `VkEndCommandBuffer` for the primary command buffer that the render pass is recorded into


// Provided by VK_VERSION_1_0
typedef VkFlags VkImageViewCreateFlags;

**VkImageViewCreateFlags** is a bitmask type for setting a mask of zero or more **VkImageViewCreateFlagBits**.

The set of usages for the created image view can be restricted compared to the parent image's usage flags by adding a **VkImageViewUsageCreateInfo** structure to the **pNext** chain of **VkImageViewCreateInfo**.

The **VkImageViewUsageCreateInfo** structure is defined as:

// Provided by VK_VERSION_1_1
typedef struct VkImageViewUsageCreateInfo {
    VkStructureType      sType;
    const void*          pNext;
    VkImageUsageFlags    usage;
} VkImageViewUsageCreateInfo;

or the equivalent

// Provided by VK_KHR_maintenance2
typedef VkImageViewUsageCreateInfo VkImageViewUsageCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **usage** is a bitmask describing the allowed usages of the image view. See **VkImageUsageFlagBits** for a description of the supported bits.

When this structure is chained to **VkImageViewCreateInfo** the **usage** field overrides the implicit **usage** parameter inherited from image creation time and its value is used instead for the purposes of determining the valid usage conditions of **VkImageViewCreateInfo**.

### Valid Usage (Implicit)

- **VUID-VkImageViewUsageCreateInfo-sType-sType** 
  **sType** must be **VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO**

- **VUID-VkImageViewUsageCreateInfo-usage-parameter**
  **usage** must be a valid combination of **VkImageUsageFlagBits** values

- **VUID-VkImageViewUsageCreateInfo-usage-requiredbitmask**
  **usage** must not be **0**

The **VkImageSubresourceRange** structure is defined as:
// Provided by VK_VERSION_1_0

typedef struct VkImageSubresourceRange {
    VkImageAspectFlags aspectMask;
    uint32_t baseMipLevel;
    uint32_t levelCount;
    uint32_t baseArrayLayer;
    uint32_t layerCount;
} VkImageSubresourceRange;

• `aspectMask` is a bitmask of `VkImageAspectFlagBits` specifying which aspect(s) of the image are included in the view.
• `baseMipLevel` is the first mipmap level accessible to the view.
• `levelCount` is the number of mipmap levels (starting from `baseMipLevel`) accessible to the view.
• `baseArrayLayer` is the first array layer accessible to the view.
• `layerCount` is the number of array layers (starting from `baseArrayLayer`) accessible to the view.

The number of mipmap levels and array layers must be a subset of the image subresources in the image. If an application wants to use all mip levels or layers in an image after the `baseMipLevel` or `baseArrayLayer`, it can set `levelCount` and `layerCount` to the special values `VK_REMAINING_MIP_LEVELS` and `VK_REMAINING_ARRAY_LAYERS` without knowing the exact number of mip levels or layers.

For cube and cube array image views, the layers of the image view starting at `baseArrayLayer` correspond to faces in the order +X, -X, +Y, -Y, +Z, -Z. For cube arrays, each set of six sequential layers is a single cube, so the number of cube maps in a cube map array view is `layerCount / 6`, and image array layer (`baseArrayLayer + i`) is face index `(i mod 6)` of cube `i / 6`. If the number of layers in the view, whether set explicitly in `layerCount` or implied by `VK_REMAINING_ARRAY_LAYERS`, is not a multiple of 6, the last cube map in the array must not be accessed.

`aspectMask` must be only `VK_IMAGE_ASPECT_COLOR_BIT`, `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` if `format` is a color, depth-only or stencil-only format, respectively, except if `format` is a multi-planar format. If using a depth/stencil format with both depth and stencil components, `aspectMask` must include at least one of `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT`, and can include both.

When the `VkImageSubresourceRange` structure is used to select a subset of the slices of a 3D image's mip level in order to create a 2D or 2D array image view of a 3D image created with `VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT`, `baseArrayLayer` and `layerCount` specify the first slice index and the number of slices to include in the created image view. Such an image view can be used as a framebuffer attachment that refers only to the specified range of slices of the selected mip level. However, any layout transitions performed on such an attachment view during a render pass instance still apply to the entire subresource referenced which includes all the slices of the selected mip level.

When using an image view of a depth/stencil image to populate a descriptor set (e.g. for sampling in the shader, or for use as an input attachment), the `aspectMask` must only include one bit and selects whether the image view is used for depth reads (i.e. using a floating-point sampler or input attachment in the shader) or stencil reads (i.e. using an unsigned integer sampler or input
attachment in the shader). When an image view of a depth/stencil image is used as a depth/stencil framebuffer attachment, the aspectMask is ignored and both depth and stencil image subresources are used.

When creating a VkImageView, if sampler Y’CbCr conversion is enabled in the sampler, the aspectMask of a subresourceRange used by the VkImageView must be VK_IMAGE_ASPECT_COLOR_BIT.

When creating a VkImageView, if sampler Y’CbCr conversion is not enabled in the sampler and the image format is multi-planar, the image must have been created with VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT, and the aspectMask of the VkImageView’s subresourceRange must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT.

Valid Usage

• VUID-VkImageSubresourceRange-levelCount-01720
  If levelCount is not VK_REMAINING_MIP_LEVELS, it must be greater than 0

• VUID-VkImageSubresourceRange-layerCount-01721
  If layerCount is not VK_REMAINING_ARRAY_LAYERS, it must be greater than 0

• VUID-VkImageSubresourceRange-aspectMask-01670
  If aspectMask includes VK_IMAGE_ASPECT_COLOR_BIT, then it must not include any of VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

• VUID-VkImageSubresourceRange-aspectMask-02278
  aspectMask must not include VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT for any index i

Valid Usage (Implicit)

• VUID-VkImageSubresourceRange-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values

• VUID-VkImageSubresourceRange-aspectMask-requiredbitmask
  aspectMask must not be 0

Bits which can be set in an aspect mask to specify aspects of an image for purposes such as identifying a subresource, are:
typedef enum VkImageAspectFlagBits {
    VK_IMAGE_ASPECT_COLOR_BIT = 0x00000001,
    VK_IMAGE_ASPECT_DEPTH_BIT = 0x00000002,
    VK_IMAGE_ASPECT_STENCIL_BIT = 0x00000004,
    VK_IMAGE_ASPECT_METADATA_BIT = 0x00000008,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_0_BIT = 0x00000010,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_1_BIT = 0x00000020,
    // Provided by VK_VERSION_1_1
    VK_IMAGE_ASPECT_PLANE_2_BIT = 0x00000040,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT = 0x00000080,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT = 0x00000100,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT = 0x00000200,
    // Provided by VK_EXT_image_drm_format_modifier
    VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT = 0x00000400,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_IMAGE_ASPECT_PLANE_0_BIT_KHR = VK_IMAGE_ASPECT_PLANE_0_BIT,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_IMAGE_ASPECT_PLANE_1_BIT_KHR = VK_IMAGE_ASPECT_PLANE_1_BIT,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_IMAGE_ASPECT_PLANE_2_BIT_KHR = VK_IMAGE_ASPECT_PLANE_2_BIT,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_IMAGE_ASPECT_PLANE_2_BIT_KHR = VK_IMAGE_ASPECT_PLANE_2_BIT,
} VkImageAspectFlagBits;

- **VK_IMAGE_ASPECT_COLOR_BIT** specifies the color aspect.
- **VK_IMAGE_ASPECT_DEPTH_BIT** specifies the depth aspect.
- **VK_IMAGE_ASPECT_STENCIL_BIT** specifies the stencil aspect.
- **VK_IMAGE_ASPECT_METADATA_BIT** specifies the metadata aspect, used for sparse sparse resource operations.
- **VK_IMAGE_ASPECT_PLANE_0_BIT** specifies plane 0 of a multi-planar image format.
- **VK_IMAGE_ASPECT_PLANE_1_BIT** specifies plane 1 of a multi-planar image format.
- **VK_IMAGE_ASPECT_PLANE_2_BIT** specifies plane 2 of a multi-planar image format.
- **VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT** specifies memory plane 0.
- **VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT** specifies memory plane 1.
- **VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT** specifies memory plane 2.
- **VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT** specifies memory plane 3.

// Provided by VK_VERSION_1_0
typedef VkFlags VkImageAspectFlags;
**VkImageAspectFlags** is a bitmask type for setting a mask of zero or more **VkImageAspectFlagBits**.

The **VkComponentMapping** structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkComponentMapping {
    VkComponentSwizzle r;
    VkComponentSwizzle g;
    VkComponentSwizzle b;
    VkComponentSwizzle a;
} VkComponentMapping;
```

- **r** is a **VkComponentSwizzle** specifying the component value placed in the R component of the output vector.
- **g** is a **VkComponentSwizzle** specifying the component value placed in the G component of the output vector.
- **b** is a **VkComponentSwizzle** specifying the component value placed in the B component of the output vector.
- **a** is a **VkComponentSwizzle** specifying the component value placed in the A component of the output vector.

**Valid Usage (Implicit)**

- **VUID-VkComponentMapping-r-parameter**
  - **r** must be a valid **VkComponentSwizzle** value
- **VUID-VkComponentMapping-g-parameter**
  - **g** must be a valid **VkComponentSwizzle** value
- **VUID-VkComponentMapping-b-parameter**
  - **b** must be a valid **VkComponentSwizzle** value
- **VUID-VkComponentMapping-a-parameter**
  - **a** must be a valid **VkComponentSwizzle** value

Possible values of the members of **VkComponentMapping**, specifying the component values placed in each component of the output vector, are:
typedef enum VkComponentSwizzle {
    VK_COMPONENT_SWIZZLE_IDENTITY = 0,
    VK_COMPONENT_SWIZZLE_ZERO = 1,
    VK_COMPONENT_SWIZZLE_ONE = 2,
    VK_COMPONENT_SWIZZLE_R = 3,
    VK_COMPONENT_SWIZZLE_G = 4,
    VK_COMPONENT_SWIZZLE_B = 5,
    VK_COMPONENT_SWIZZLE_A = 6,
} VkComponentSwizzle;

- **VK_COMPONENT_SWIZZLE_IDENTITY** specifies that the component is set to the identity swizzle.
- **VK_COMPONENT_SWIZZLE_ZERO** specifies that the component is set to zero.
- **VK_COMPONENT_SWIZZLE_ONE** specifies that the component is set to either 1 or 1.0, depending on whether the type of the image view format is integer or floating-point respectively, as determined by the Format Definition section for each VkFormat.
- **VK_COMPONENT_SWIZZLE_R** specifies that the component is set to the value of the R component of the image.
- **VK_COMPONENT_SWIZZLE_G** specifies that the component is set to the value of the G component of the image.
- **VK_COMPONENT_SWIZZLE_B** specifies that the component is set to the value of the B component of the image.
- **VK_COMPONENT_SWIZZLE_A** specifies that the component is set to the value of the A component of the image.

Setting the identity swizzle on a component is equivalent to setting the identity mapping on that component. That is:

<table>
<thead>
<tr>
<th>Component</th>
<th>Identity Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>components.r</td>
<td>VK_COMPONENT_SWIZZLE_R</td>
</tr>
<tr>
<td>components.g</td>
<td>VK_COMPONENT_SWIZZLE_G</td>
</tr>
<tr>
<td>components.b</td>
<td>VK_COMPONENT_SWIZZLE_B</td>
</tr>
<tr>
<td>components.a</td>
<td>VK_COMPONENT_SWIZZLE_A</td>
</tr>
</tbody>
</table>

If the pNext list includes a VkImageViewASTCDecodeModeEXT structure, then that structure includes a parameter specifying the decode mode for image views using ASTC compressed formats.

The VkImageViewASTCDecodeModeEXT structure is defined as:
typedef struct VkImageViewASTCDecodeModeEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkFormat           decodeMode;
} VkImageViewASTCDecodeModeEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **decodeMode** is the intermediate format used to decode ASTC compressed formats.

**Valid Usage**

- VUID-VkImageViewASTCDecodeModeEXT-decodeMode-02230
  decodeMode must be one of VK_FORMAT_R16G16B16A16_SFLOAT, VK_FORMAT_R8G8B8A8_UNORM, or VK_FORMAT_E5B9G9R9_UFLOAT_PACK32

- VUID-VkImageViewASTCDecodeModeEXT-decodeMode-02231
  If the decodeModeSharedExponent feature is not enabled, decodeMode must not be VK_FORMAT_E5B9G9R9_UFLOAT_PACK32

- VUID-VkImageViewASTCDecodeModeEXT-decodeMode-02232
  If decodeMode is VK_FORMAT_R8G8B8A8_UNORM the image view must not include blocks using any of the ASTC HDR modes

- VUID-VkImageViewASTCDecodeModeEXT-format-04084
  format of the image view must be one of the ASTC Compressed Image Formats

If format uses sRGB encoding then the decodeMode has no effect.

**Valid Usage (Implicit)**

- VUID-VkImageViewASTCDecodeModeEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT

- VUID-VkImageViewASTCDecodeModeEXT-decodeMode-parameter
  decodeMode must be a valid VkFormat value

To destroy an image view, call:

// Provided by VK_VERSION_1_0
void vkDestroyImageView(
    VkDevice                                    device,
    VkImageView                                 imageView,
    const VkAllocationCallbacks*                pAllocator);

- **device** is the logical device that destroys the image view.
• **imageView** is the image view to destroy.
• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyImageView-imageView-01026**
  All submitted commands that refer to **imageView** must have completed execution

- **VUID-vkDestroyImageView-imageView-01027**
  If **VkAllocationCallbacks** were provided when **imageView** was created, a compatible set of callbacks must be provided here.

- **VUID-vkDestroyImageView-imageView-01028**
  If no **VkAllocationCallbacks** were provided when **imageView** was created, **pAllocator** must be **NULL**.

### Valid Usage (Implicit)

- **VUID-vkDestroyImageView-device-parameter**
  **device** must be a valid **VkDevice** handle.

- **VUID-vkDestroyImageView-imageView-parameter**
  If **imageView** is not **VK_NULL_HANDLE**, **imageView** must be a valid **VkImageView** handle.

- **VUID-vkDestroyImageView-pAllocator-parameter**
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure.

- **VUID-vkDestroyImageView-imageView-parent**
  If **imageView** is a valid handle, it must have been created, allocated, or retrieved from **device**.

### Host Synchronization

- Host access to **imageView** must be externally synchronized.

To get the handle for an image view, call:

```c
// Provided by VK_NVX_image_view_handle
uint32_t vkGetImageViewHandleNVX(
    VkDevice device,
    const VkImageViewHandleInfoNVX* pInfo);
```

- **device** is the logical device that owns the image view.
- **pInfo** describes the image view to query and type of handle.
Valid Usage (Implicit)

- VUID-vkGetImageViewHandleNVX-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetImageViewHandleNVX-pInfo-parameter
pInfo must be a valid pointer to a valid VkImageViewHandleInfoNVX structure

The VkImageViewHandleInfoNVX structure is defined as:

```c
// Provided by VK_NVX_image_view_handle
typedef struct VkImageViewHandleInfoNVX {
    VkStructureType     sType;
    const void*         pNext;
    VkImageView         imageView;
    VkDescriptorType    descriptorType;
    VkSampler           sampler;
} VkImageViewHandleInfoNVX;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `imageView` is the image view to query.
- `descriptorType` is the type of descriptor for which to query a handle.
- `sampler` is the sampler to combine with the image view when generating the handle.

Valid Usage

- VUID-VkImageViewHandleInfoNVX-descriptorType-02654
descriptorType must be VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,
  VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER

- VUID-VkImageViewHandleInfoNVX-sampler-02655
sampler must be a valid VkSampler if descriptorType is
  VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER

- VUID-VkImageViewHandleInfoNVX-imageView-02656
If descriptorType is VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE or
  VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, the image that imageView was created from
  must have been created with the VK_IMAGE_USAGE_SAMPLED_BIT usage bit set

- VUID-VkImageViewHandleInfoNVX-imageView-02657
If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, the image that imageView was
  created from must have been created with the VK_IMAGE_USAGE_STORAGE_BIT usage bit set
Valid Usage (Implicit)

- **VUID-VkImageViewHandleInfoNVX-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_HANDLE_INFO_NVX`.

- **VUID-VkImageViewHandleInfoNVX-pNext-pNext**
  
  `pNext` must be `NULL`.

- **VUID-VkImageViewHandleInfoNVX-imageView-parameter**
  
  `imageView` must be a valid `VkImageView` handle.

- **VUID-VkImageViewHandleInfoNVX-descriptorType-parameter**
  
  `descriptorType` must be a valid `VkDescriptorType` value.

- **VUID-VkImageViewHandleInfoNVX-sampler-parameter**
  
  If `sampler` is `VK_NULL_HANDLE`, `sampler` must be a valid `VkSampler` handle.

- **VUID-VkImageViewHandleInfoNVX-commonparent**
  
  Both of `imageView`, and `sampler` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

To get the device address for an image view, call:

```c
// Provided by VK_NVX_image_view_handle
VkResult vkGetImageViewAddressNVX(
    VkDevice                                    device,
    VkImageView                                 imageView,
    VkImageViewAddressPropertiesNVX*            pProperties);
```

- `device` is the logical device that owns the image view.
- `imageView` is a handle to the image view.
- `pProperties` contains the device address and size when the call returns.

Valid Usage (Implicit)

- **VUID-vkGetImageViewAddressNVX-device-parameter**
  
  `device` must be a valid `VkDevice` handle.

- **VUID-vkGetImageViewAddressNVX-imageView-parameter**
  
  `imageView` must be a valid `VkImageView` handle.

- **VUID-vkGetImageViewAddressNVX-pProperties-parameter**
  
  `pProperties` must be a valid pointer to a `VkImageViewAddressPropertiesNVX` structure.

- **VUID-vkGetImageViewAddressNVX-imageView-parent**
  
  `imageView` must have been created, allocated, or retrieved from `device`.
Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_UNKNOWN**

The `VkImageViewAddressPropertiesNVX` structure is defined as:

```c
// Provided by VK_NVX_image_view_handle
typedef struct VkImageViewAddressPropertiesNVX {
    VkStructureType    sType;
    void*              pNext;
    VkDeviceAddress    deviceAddress;
    VkDeviceSize       size;
} VkImageViewAddressPropertiesNVX;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `deviceAddress` is the device address of the image view.
- `size` is the size in bytes of the image view device memory.

**Valid Usage (Implicit)**

- **VUID-VkImageViewAddressPropertiesNVX-sType-sType**  
  `sType` **must be** `VK_STRUCTURE_TYPE_IMAGE_VIEW_ADDRESS_PROPERTIES_NVX`
- **VUID-VkImageViewAddressPropertiesNVX-pNext-pNext**  
  `pNext` **must be** `NULL`

### 12.5.1. Image View Format Features

Valid uses of a `VkImageView` may depend on the image view’s `format features`, defined below. Such constraints are documented in the affected valid usage statement.

- If `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_LINEAR`, then the image view’s set of `format features` is the value of `VkFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` on the same `format` as `VkImageViewCreateInfo::format`.

- If `VkImageViewCreateInfo::image` was created with `VK_IMAGE_TILING_OPTIMAL`, but without an Android hardware buffer external format, then the image view’s set of `format features` is the value of `VkFormatProperties::optimalTilingFeatures` found by calling
vkGetPhysicalDeviceFormatProperties on the same format as VkImageViewCreateInfo::format.

- If VkImageViewCreateInfo::image was created with an Android hardware buffer external format, then the image views's set of format features is the value of VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures found by calling vkGetAndroidHardwareBufferPropertiesANDROID on the Android hardware buffer that was imported to the VkDeviceMemory to which the VkImageViewCreateInfo::image is bound.

- If VkImageViewCreateInfo::image was created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then:
  - The image's DRM format modifier is the value of VkImageDrmFormatModifierListCreateInfoEXT::drmFormatModifier found by calling vkGetImageDrmFormatModifierPropertiesEXT.
  - Let VkDrmFormatModifierPropertiesListEXT::pDrmFormatModifierProperties be the array found by calling vkGetPhysicalDeviceFormatProperties2 on the same format as VkImageViewCreateInfo::format.
  - Let VkDrmFormatModifierPropertiesEXT prop be an array element whose drmFormatModifier member is the value of the image's DRM format modifier.
  - Then the image view's set of format features is the value of taking the bitwise intersection, over the collected prop::drmFormatModifierTilingFeatures.

### 12.6. Acceleration Structures

Acceleration structures are an opaque structure that is built by the implementation to more efficiently perform spatial queries on the provided geometric data. For this extension, an acceleration structure is either a top-level acceleration structure containing a set of bottom-level acceleration structures or a bottom-level acceleration structure containing either a set of axis-aligned bounding boxes for custom geometry or a set of triangles.

Each instance in the top-level acceleration structure contains a reference to a bottom-level acceleration structure as well as an instance transform plus information required to index into the shader bindings. The top-level acceleration structure is what is bound to the acceleration descriptor, for example to trace inside the shader in the ray tracing pipeline.

Acceleration structures are represented by VkAccelerationStructureKHR handles:

```c
// Provided by VK_KHR_acceleration_structure
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkAccelerationStructureKHR)
```

Acceleration structures for the VK_NV_ray_tracing extension are represented by the similar VkAccelerationStructureNV handles:

```c
// Provided by VK_NV_ray_tracing
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkAccelerationStructureNV)
```

To create acceleration structures, call:
// Provided by VK_NV_ray_tracing
VkResult vkCreateAccelerationStructureNV(
    VkDevice                                    device,
    const VkAccelerationStructureCreateInfoNV*  pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkAccelerationStructureNV*                  pAccelerationStructure);

• **device** is the logical device that creates the buffer object.

• **pCreateInfo** is a pointer to a `VkAccelerationStructureCreateInfoNV` structure containing parameters affecting creation of the acceleration structure.

• **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

• **pAccelerationStructure** is a pointer to a `VkAccelerationStructureNV` handle in which the resulting acceleration structure object is returned.

Similar to other objects in Vulkan, the acceleration structure creation merely creates an object with a specific “shape” as specified by the information in `VkAccelerationStructureInfoNV` and `compactedSize` in `pCreateInfo`. Populating the data in the object after allocating and binding memory is done with `vkCmdBuildAccelerationStructureNV` and `vkCmdCopyAccelerationStructureNV`.

Acceleration structure creation uses the count and type information from the geometries, but does not use the data references in the structures.

### Valid Usage (Implicit)

- **VUID-vkCreateAccelerationStructureNV-device-parameter**  
  
  **device** must be a valid `VkDevice` handle

- **VUID-vkCreateAccelerationStructureNV-pCreateInfo-parameter**  
  
  **pCreateInfo** must be a valid pointer to a valid `VkAccelerationStructureCreateInfoNV` structure

- **VUID-vkCreateAccelerationStructureNV-pAllocator-parameter**  
  
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateAccelerationStructureNV-pAccelerationStructure-parameter**  
  
  **pAccelerationStructure** must be a valid pointer to a `VkAccelerationStructureNV` handle

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
The `VkAccelerationStructureCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkAccelerationStructureCreateInfoNV {
    VkStructureType                  sType;
    const void*                      pNext;
    VkDeviceSize                     compactedSize;
    VkAccelerationStructureInfoNV    info;
} VkAccelerationStructureCreateInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **compactedSize** is the size from the result of `vkCmdWriteAccelerationStructuresPropertiesNV` if this acceleration structure is going to be the target of a compacting copy.
- **info** is the `VkAccelerationStructureInfoNV` structure specifying further parameters of the created acceleration structure.

**Valid Usage**

- VUID-VkAccelerationStructureCreateInfoNV-compactedSize-02421
  If `compactedSize` is not 0 then both `info.geometryCount` and `info.instanceCount` must be 0

**Valid Usage (Implicit)**

- VUID-VkAccelerationStructureCreateInfoNV-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV`
- VUID-VkAccelerationStructureCreateInfoNV-pNext-pNext
  `pNext` must be NULL
- VUID-VkAccelerationStructureCreateInfoNV-info-parameter
  `info` must be a valid `VkAccelerationStructureInfoNV` structure

The `VkAccelerationStructureInfoNV` structure is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkAccelerationStructureInfoNV {
    VkStructureType                        sType;
    const void*                            pNext;
    VkAccelerationStructureTypeNV          type;
    VkBuildAccelerationStructureFlagsNV    flags;
    uint32_t                               instanceCount;
    uint32_t                               geometryCount;
    const VkGeometryNV*                    pGeometries;
} VkAccelerationStructureInfoNV;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **type** is a `VkAccelerationStructureTypeNV` value specifying the type of acceleration structure that will be created.

• **flags** is a bitmask of `VkBuildAccelerationStructureFlagBitsNV` specifying additional parameters of the acceleration structure.

• **instanceCount** specifies the number of instances that will be in the new acceleration structure.

• **geometryCount** specifies the number of geometries that will be in the new acceleration structure.

• **pGeometries** is a pointer to an array of `geometryCount` `VkGeometryNV` structures containing the scene data being passed into the acceleration structure.

`VkAccelerationStructureInfoNV` contains information that is used both for acceleration structure creation with `vkCreateAccelerationStructureNV` and in combination with the actual geometric data to build the acceleration structure with `vkCmdBuildAccelerationStructureNV`. 
Valid Usage

- **VUID-VkAccelerationStructureInfoNV-geometryCount-02422**
  
  `geometryCount` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxGeometryCount`

- **VUID-VkAccelerationStructureInfoNV-instanceCount-02423**
  
  `instanceCount` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxInstanceCount`

- **VUID-VkAccelerationStructureInfoNV-maxTriangleCount-02424**
  
  The total number of triangles in all geometries **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxTriangleCount`

- **VUID-VkAccelerationStructureInfoNV-type-02425**
  
  If `type` is `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV` then `geometryCount` **must** be 0

- **VUID-VkAccelerationStructureInfoNV-type-02426**
  
  If `type` is `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV` then `instanceCount` **must** be 0

- **VUID-VkAccelerationStructureInfoNV-type-02786**
  
  If `type` is `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV` then the `geometryType` member of each geometry in `pGeometries` **must** be the same

- **VUID-VkAccelerationStructureInfoNV-type-04623**
  
  `type` **must** not be `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR`

- **VUID-VkAccelerationStructureInfoNV-flags-02592**
  
  If `flags` has the `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV` bit set, then it **must** not have the `VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_NV` bit set

- **VUID-VkAccelerationStructureInfoNV-scratch-02781**
  
  `scratch` **must** have been created with `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` usage flag

- **VUID-VkAccelerationStructureInfoNV-instanceData-02782**
  
  If `instanceData` is not `VK_NULL_HANDLE`, `instanceData` **must** have been created with `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` usage flag
Valid Usage (Implicit)

- **VUID-VkAccelerationStructureInfoNV-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_INFO_NV`

- **VUID-VkAccelerationStructureInfoNV-pNext-pNext**
  - `pNext` must be `NULL`

- **VUID-VkAccelerationStructureInfoNV-type-parameter**
  - `type` must be a valid `VkAccelerationStructureTypeNV` value

- **VUID-VkAccelerationStructureInfoNV-flags-parameter**
  - `flags` must be a valid combination of `VkBuildAccelerationStructureFlagBitsNV` values

- **VUID-VkAccelerationStructureInfoNV-pGeometries-parameter**
  - If `geometryCount` is not 0, `pGeometries` must be a valid pointer to an array of `geometryCount` valid `VkGeometryNV` structures

To create an acceleration structure, call:

```c
// Provided by VK_KHR_acceleration_structure
VkResult vkCreateAccelerationStructureKHR(
    VkDevice                                    device,
    const VkAccelerationStructureCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkAccelerationStructureKHR*                 pAccelerationStructure);
```

- **device** is the logical device that creates the acceleration structure object.

- **pCreateInfo** is a pointer to a `VkAccelerationStructureCreateInfoKHR` structure containing parameters affecting creation of the acceleration structure.

- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

- **pAccelerationStructure** is a pointer to a `VkAccelerationStructureKHR` handle in which the resulting acceleration structure object is returned.

Similar to other objects in Vulkan, the acceleration structure creation merely creates an object with a specific “shape”. The type and quantity of geometry that can be built into an acceleration structure is determined by the parameters of `VkAccelerationStructureCreateInfoKHR`.

Populating the data in the object after allocating and binding memory is done with commands such as `vkCmdBuildAccelerationStructuresKHR`, `vkBuildAccelerationStructuresKHR`, `vkCmdCopyAccelerationStructureKHR`, and `vkCopyAccelerationStructureKHR`.

The input buffers passed to acceleration structure build commands will be referenced by the implementation for the duration of the command. After the command completes, the acceleration structure may hold a reference to any acceleration structure specified by an active instance contained therein. Apart from this referencing, acceleration structures must be fully self-contained. The application may re-use or free any memory which was used by the command as an input or as scratch without affecting the results of ray traversal.
Valid Usage

- VUID-vkCreateAccelerationStructureKHR-accelerationStructure-03611
  The `accelerationStructure` feature must be enabled

- VUID-vkCreateAccelerationStructureKHR-deviceAddress-03488
  If `VkAccelerationStructureCreateInfoKHR::deviceAddress` is not zero, the `accelerationStructureCaptureReplay` feature must be enabled

- VUID-vkCreateAccelerationStructureKHR-device-03489
  If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature must be enabled

Valid Usage (Implicit)

- VUID-vkCreateAccelerationStructureKHR-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkCreateAccelerationStructureKHR-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkAccelerationStructureCreateInfoKHR` structure

- VUID-vkCreateAccelerationStructureKHR-pAllocator-parameter
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateAccelerationStructureKHR-pAccelerationStructure-parameter
  `pAccelerationStructure` must be a valid pointer to a `VkAccelerationStructureKHR` handle

Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

The `VkAccelerationStructureCreateInfoKHR` structure is defined as:
typedef struct VkAccelerationStructureCreateInfoKHR {  
  VkStructureType                          sType;  
  const void*                              pNext;  
  VkAccelerationStructureCreateFlagsKHR    createFlags;  
  VkBuffer                                 buffer;  
  VkDeviceSize                             offset;  
  VkDeviceSize                             size;  
  VkAccelerationStructureTypeKHR           type;  
  VkDeviceAddress                          deviceAddress;  
} VkAccelerationStructureCreateInfoKHR;

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **createFlags** is a bitmask of **VkAccelerationStructureCreateFlagBitsKHR** specifying additional creation parameters of the acceleration structure.

• **buffer** is the buffer on which the acceleration structure will be stored.

• **offset** is an offset in bytes from the base address of the buffer at which the acceleration structure will be stored, and **must** be a multiple of **256**.

• **size** is the size required for the acceleration structure.

• **type** is a **VkAccelerationStructureTypeKHR** value specifying the type of acceleration structure that will be created.

• **deviceAddress** is the device address requested for the acceleration structure if the **accelerationStructureCaptureReplay** feature is being used.

If **deviceAddress** is zero, no specific address is requested.

If **deviceAddress** is not zero, **deviceAddress** **must** be an address retrieved from an identically created acceleration structure on the same implementation. The acceleration structure **must** also be placed on an identically created **buffer** and at the same **offset**.

Applications **should** avoid creating acceleration structures with application-provided addresses and implementation-provided addresses in the same process, to reduce the likelihood of **VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR** errors.
Note

The expected usage for this is that a trace capture/replay tool will add the `VK_BUFFER_CREATEDEVICE_ADDRESS_CAPTURE_REPLAY_BIT` flag to all buffers that use `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT`, and `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT` to all buffers used as storage for an acceleration structure where `deviceAddress` is not zero. During capture the tool will save the queried opaque device addresses in the trace. During replay, the buffers will be created specifying the original address so any address values stored in the trace data will remain valid.

Implementations are expected to separate such buffers in the GPU address space so normal allocations will avoid using these addresses. Apps/tools should avoid mixing app-provided and implementation-provided addresses for buffers created with `VK_BUFFER_CREATEDEVICE_ADDRESS_CAPTURE_REPLAY_BIT`, to avoid address space allocation conflicts.

Applications should create an acceleration structure with a specific `VkAccelerationStructureTypeKHR` other than `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR`.

If the acceleration structure will be the target of a build operation, the required size for an acceleration structure can be queried with `vkGetAccelerationStructureBuildSizesKHR`. If the acceleration structure is going to be the target of a compacting copy, `vkCmdWriteAccelerationStructuresPropertiesKHR` or `vkWriteAccelerationStructuresPropertiesKHR` can be used to obtain the compacted size required.

Valid Usage

- VUID-VkAccelerationStructureCreateInfoKHR-deviceAddress-03612
  If `deviceAddress` is not zero, `createFlags` must include `VK_ACCELERATION_STRUCTURE_CREATEDEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`

- VUID-VkAccelerationStructureCreateInfoKHR-createFlags-03613
  If `createFlags` includes `VK_ACCELERATION_STRUCTURE_CREATEDEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR`, `VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureCaptureReplay` must be `VK_TRUE`

- VUID-VkAccelerationStructureCreateInfoKHR-buffer-03614
  `buffer` must have been created with a usage value containing `VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR`

- VUID-VkAccelerationStructureCreateInfoKHR-buffer-03615
  `buffer` must not have been created with `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`

- VUID-VkAccelerationStructureCreateInfoKHR-offset-03616
  The sum of `offset` and `size` must be less than the size of `buffer`

- VUID-VkAccelerationStructureCreateInfoKHR-offset-03734
  `offset` must be a multiple of 256 bytes
Valid Usage (Implicit)

- **sType must** be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR`
- **pNext must** be `NULL`
- **createFlags must** be a valid combination of `VkAccelerationStructureCreateFlagBitsKHR` values
- **buffer must** be a valid `VkBuffer` handle
- **type must** be a valid `VkAccelerationStructureTypeKHR` value

To get the build sizes for an acceleration structure, call:

```c
// Provided by VK_KHR_acceleration_structure
void vkGetAccelerationStructureBuildSizesKHR(
    VkDevice                                    device,
    VkAccelerationStructureBuildTypeKHR         buildType,
    const          VkAccelerationStructureBuildGeometryInfoKHR* pBuildInfo,
    const          uint32_t*                             pMaxPrimitiveCounts,
    VkAccelerationStructureBuildSizesInfoKHR*   pSizeInfo);
```

- **device** is the logical device that will be used for creating the acceleration structure.
- **buildType** defines whether host or device operations (or both) are being queried for.
- **pBuildInfo** is a pointer to a `VkAccelerationStructureBuildGeometryInfoKHR` structure describing parameters of a build operation.
- **pMaxPrimitiveCounts** is a pointer to an array of `pBuildInfo->geometryCount` `uint32_t` values defining the number of primitives built into each geometry.
- **pSizeInfo** is a pointer to a `VkAccelerationStructureBuildSizesInfoKHR` structure which returns the size required for an acceleration structure and the sizes required for the scratch buffers, given the build parameters.

The `srcAccelerationStructure`, `dstAccelerationStructure`, and `mode` members of `pBuildInfo` are ignored. Any `VkDeviceOrHostAddressKHR` members of `pBuildInfo` are ignored by this command, except that the `hostAddress` member of `VkAccelerationStructureGeometryTrianglesDataKHR::transformData` will be examined to check if it is `NULL`.

An acceleration structure created with the `accelerationStructureSize` returned by this command supports any build or update with a `VkAccelerationStructureBuildGeometryInfoKHR` structure and array of `VkAccelerationStructureBuildRangeInfoKHR` structures subject to the following properties:

- The `build` command is a host build command, and `buildType` is
VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_KHR or
VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_OR_DEVICE_KHR

• The build command is a device build command, and buildType is
  VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR or
  VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_OR_DEVICE_KHR

• For VkAccelerationStructureBuildGeometryInfoKHR:
  ◦ Its type, and flags members are equal to those specified in pBuildInfo.
  ◦ geometryCount is less than or equal to that specified in pBuildInfo.
  ◦ For each element of either pGeometries or ppGeometries at a given index, its geometryType member is equal to that specified in pBuildInfo.
  ◦ For each element of either pGeometries or ppGeometries at a given index, with a geometryType member equal to VK_GEOMETRY_TYPE_TRIANGLES_KHR, the vertexFormat and indexType members of geometry.triangles are equal to those specified in the same element in pBuildInfo.
  ◦ For each element of either pGeometries or ppGeometries at a given index, with a geometryType member equal to VK_GEOMETRY_TYPE_TRIANGLES_KHR, the maxVertex member of geometry.triangles is less than or equal to that specified in the same element in pBuildInfo.
  ◦ For each element of either pGeometries or ppGeometries at a given index, with a geometryType member equal to VK_GEOMETRY_TYPE_TRIANGLES_KHR, if the applicable address in the transformData member of geometry.triangles is not NULL, the corresponding transformData.hostAddress parameter in pBuildInfo is not NULL.

  • For each VkAccelerationStructureBuildRangeInfoKHR corresponding to the
    VkAccelerationStructureBuildGeometryInfoKHR:
    ◦ Its primitiveCount member is less than or equal to the corresponding element of pMaxPrimitiveCounts.

Similarly, the updateScratchSize value will support any build command specifying the
VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR mode under the above conditions, and the
buildScratchSize value will support any build command specifying the
VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR mode under the above conditions.
Valid Usage

- VUID-vkGetAccelerationStructureBuildSizesKHR-rayTracingPipeline-03617
  The rayTracingPipeline or rayQuery feature must be enabled

- VUID-vkGetAccelerationStructureBuildSizesKHR-device-03618
  If device was created with multiple physical devices, then the
  bufferDeviceAddressMultiDevice feature must be enabled

- VUID-vkGetAccelerationStructureBuildSizesKHR-pBuildInfo-03619
  If pBuildInfo->geometryCount is not 0, pMaxPrimitiveCounts must be a valid pointer to an
  array of pBuildInfo->geometryCount uint32_t values

- VUID-vkGetAccelerationStructureBuildSizesKHR-pBuildInfo-03785
  If pBuildInfo->pGeometries or pBuildInfo->ppGeometries has a geometryType of
  VK_GEOMETRY_TYPE_INSTANCES_KHR, each pMaxPrimitiveCounts[i] must be less than or equal to
  VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxInstanceCount

Valid Usage (Implicit)

- VUID-vkGetAccelerationStructureBuildSizesKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetAccelerationStructureBuildSizesKHR-buildType-parameter
  buildType must be a valid VkAccelerationStructureBuildTypeKHR value

- VUID-vkGetAccelerationStructureBuildSizesKHR-pBuildInfo-parameter
  pBuildInfo must be a valid pointer to a valid
  VkAccelerationStructureBuildGeometryInfoKHR structure

- VUID-vkGetAccelerationStructureBuildSizesKHR-pMaxPrimitiveCounts-parameter
  If pMaxPrimitiveCounts is not NULL, pMaxPrimitiveCounts must be a valid pointer to an array
  of pBuildInfo->geometryCount uint32_t values

- VUID-vkGetAccelerationStructureBuildSizesKHR-pSizeInfo-parameter
  pSizeInfo must be a valid pointer to a VkAccelerationStructureBuildSizesInfoKHR structure

The VkAccelerationStructureBuildSizesInfoKHR structure describes the required build sizes for an
acceleration structure and scratch buffers and is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureBuildSizesInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize accelerationStructureSize;
    VkDeviceSize updateScratchSize;
    VkDeviceSize buildScratchSize;
} VkAccelerationStructureBuildSizesInfoKHR;
```
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **accelerationStructureSize** is the size in bytes required in a `VkAccelerationStructureKHR` for a build or update operation.
• **updateScratchSize** is the size in bytes required in a scratch buffer for an update operation.
• **buildScratchSize** is the size in bytes required in a scratch buffer for a build operation.

### Valid Usage (Implicit)

- **VUID-VkAccelerationStructureBuildSizesInfoKHR-sType-sType**
  - `sType` **must be** `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR`
- **VUID-VkAccelerationStructureBuildSizesInfoKHR-pNext-pNext**
  - `pNext` **must be** **NULL**

Values which **can be set** in `VkAccelerationStructureCreateInfoKHR::type` or `VkAccelerationStructureInfoNV::type` specifying the type of acceleration structure, are:

```c
// Provided by VK_KHR_acceleration_structure
typedef enum VkAccelerationStructureTypeKHR {
    VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR = 0,
    VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR = 1,
    VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR = 2,
    // Provided by VK_NV_ray_tracing
    VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV =
    VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR,
    // Provided by VK_NV_ray_tracing
    VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV =
    VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR,
} VkAccelerationStructureTypeKHR;
```

or the equivalent

```c
// Provided by VK_NV_ray_tracing
typedef VkAccelerationStructureTypeKHR VkAccelerationStructureTypeNV;
```

- **VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR** is a top-level acceleration structure containing instance data referring to bottom-level acceleration structures.
- **VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR** is a bottom-level acceleration structure containing the AABBs or geometry to be intersected.
- **VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR** is an acceleration structure whose type is determined at build time used for special circumstances.

**Bits which can be set** in `VkAccelerationStructureCreateInfoKHR::createFlags` specifying additional
creation parameters for acceleration structures, are:

```c
// Provided by VK_KHR_acceleration_structure
typedef enum VkAccelerationStructureCreateFlagBitsKHR {
    VK_ACCELERATION_STRUCTURE_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR = 0x00000001,
} VkAccelerationStructureCreateFlagBitsKHR;
```

- **VK_ACCELERATION_STRUCTURE_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR** specifies that the acceleration structure's address can be saved and reused on a subsequent run.

```c
// Provided by VK_KHR_acceleration_structure
typedef VkFlags VkAccelerationStructureCreateFlagsKHR;
```

`VkAccelerationStructureCreateFlagsKHR` is a bitmask type for setting a mask of zero or more `VkAccelerationStructureCreateFlagBitsKHR`.

Bits which can be set in `VkAccelerationStructureBuildGeometryInfoKHR::flags` or `VkAccelerationStructureInfoNV::flags` specifying additional parameters for acceleration structure builds, are:

```c
// Provided by VK_KHR_acceleration_structure
typedef enum VkBuildAccelerationStructureFlagBitsKHR {
    VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR = 0x00000001,
    VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR = 0x00000002,
    VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR = 0x00000004,
    VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_KHR = 0x00000008,
    VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_KHR = 0x00000010,
    // Provided by VK_NV_ray_tracing
    VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV = VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_BUILD_ACCELERATION_STRUCTURE.Allow_COMPACTION_BIT_NV = VK_BUILD_ACCELERATION_STRUCTURE.Allow_COMPACTION_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV = VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_NV = VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_NV = VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_KHR,
} VkBuildAccelerationStructureFlagBitsKHR;
```

or the equivalent
 typedef VkBuildAccelerationStructureFlagBitsKHR
VkBuildAccelerationStructureFlagBitsNV;

• **VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR** indicates that the specified acceleration structure can be updated with update of *VK_TRUE* in `vkCmdBuildAccelerationStructuresKHR` or `vkCmdBuildAccelerationStructureNV`.

• **VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR** indicates that the specified acceleration structure can act as the source for a copy acceleration structure command with mode of **VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR** to produce a compacted acceleration structure.

• **VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR** indicates that the given acceleration structure build should prioritize trace performance over build time.

• **VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_KHR** indicates that the given acceleration structure build should prioritize build time over trace performance.

• **VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_KHR** indicates that this acceleration structure should minimize the size of the scratch memory and the final result build, potentially at the expense of build time or trace performance.

  
  
  **Note**

  **VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR** and **VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR** may take more time and memory than a normal build, and so should only be used when those features are needed.


 typedef VkFlags VkBuildAccelerationStructureFlagsKHR;

 or the equivalent

```
// Provided by VK_NV_ray_tracing
typedef VkBuildAccelerationStructureFlagsKHR VkBuildAccelerationStructureFlagsNV;
```

**VkBuildAccelerationStructureFlagsKHR** is a bitmask type for setting a mask of zero or more **VkBuildAccelerationStructureFlagBitsKHR**.

The **VkGeometryNV** structure describes geometry in a bottom-level acceleration structure and is defined as:
typedef struct VkGeometryNV {
    VkStructureType            sType;
    const void*                pNext;
    VkGeometryTypeKHR          geometryType;
    VkGeometryDataNV           geometry;
    VkGeometryFlagsKHR         flags;
} VkGeometryNV;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **geometryType** specifies the VkGeometryTypeKHR which this geometry refers to.
- **geometry** contains the geometry data as described in VkGeometryDataNV.
- **flags** has VkGeometryFlagBitsKHR describing options for this geometry.

### Valid Usage

- VUID-VkGeometryNV-geometryType-03503
  geometryType must be VK_GEOMETRY_TYPE_TRIANGLES_NV or VK_GEOMETRY_TYPE_AABBS_NV

### Valid Usage (Implicit)

- VUID-VkGeometryNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_GEOMETRY_NV
- VUID-VkGeometryNV-pNext-pNext
  pNext must be NULL
- VUID-VkGeometryNV-geometryType-parameter
  geometryType must be a valid VkGeometryTypeKHR value
- VUID-VkGeometryNV-geometry-parameter
  geometry must be a valid VkGeometryDataNV structure
- VUID-VkGeometryNV-flags-parameter
  flags must be a valid combination of VkGeometryFlagBitsKHR values

Geometry types are specified by VkGeometryTypeKHR, which takes values:
typedef enum VkGeometryTypeKHR {
    VK_GEOMETRY_TYPE_TRIANGLES_KHR = 0,
    VK_GEOMETRY_TYPE_AABBS_KHR = 1,
    VK_GEOMETRY_TYPE_INSTANCES_KHR = 2,
} VkGeometryTypeKHR;

or the equivalent

typedef VkGeometryTypeKHR VkGeometryTypeNV;

- **VK_GEOMETRY_TYPE_TRIANGLES_KHR** specifies a geometry type consisting of triangles.
- **VK_GEOMETRY_TYPE_AABBS_KHR** specifies a geometry type consisting of axis-aligned bounding boxes.
- **VK_GEOMETRY_TYPE_INSTANCES_KHR** specifies a geometry type consisting of acceleration structure instances.

Bits specifying additional parameters for geometries in acceleration structure builds, are:

typedef enum VkGeometryFlagBitsKHR {
    VK_GEOMETRY_OPAQUE_BIT_KHR = 0x00000001,
    VK_GEOMETRY_NO_DUPLICATE_ANY_HIT_INVOCATION_BIT_KHR = 0x00000002,
} VkGeometryFlagBitsKHR;

or the equivalent

typedef VkGeometryFlagBitsKHR VkGeometryFlagBitsNV;

- **VK_GEOMETRY_OPAQUE_BIT_KHR** indicates that this geometry does not invoke the any-hit shaders even if present in a hit group.
- **VK_GEOMETRY_NO_DUPLICATE_ANY_HIT_INVOCATION_BIT_KHR** indicates that the implementation **must** only call the any-hit shader a single time for each primitive in this geometry. If this bit is absent an implementation **may** invoke the any-hit shader more than once for this geometry.
// Provided by VK_KHR_acceleration_structure
typedef VkFlags VkGeometryFlagsKHR;

or the equivalent

// Provided by VK_NV_ray_tracing
typedef VkGeometryFlagsKHR VkGeometryFlagsNV;

VkGeometryFlagsKHR is a bitmask type for setting a mask of zero or more VkGeometryFlagBitsKHR.

The VkGeometryDataNV structure specifies geometry in a bottom-level acceleration structure and is defined as:

// Provided by VK_NV_ray_tracing
typedef struct VkGeometryDataNV {
    VkGeometryTrianglesNV triangles;
    VkGeometryAABBNV aabbs;
} VkGeometryDataNV;

- triangles contains triangle data if VkGeometryNV::geometryType is VK_GEOMETRY_TYPE_TRIANGLES_NV.
- aabbs contains axis-aligned bounding box data if VkGeometryNV::geometryType is VK_GEOMETRY_TYPE_AABBS_NV.

Valid Usage (Implicit)

- VUID-VkGeometryDataNV-triangles-parameter
  triangles must be a valid VkGeometryTrianglesNV structure
- VUID-VkGeometryDataNV-aabbs-parameter
  aabbs must be a valid VkGeometryAABBNV structure

The VkGeometryTrianglesNV structure specifies triangle geometry in a bottom-level acceleration structure and is defined as:
typedef struct VkGeometryTrianglesNV {
    VkStructureType sType;
    const void* pNext;
    VkBuffer vertexData;
    VkDeviceSize vertexOffset;
    uint32_t vertexCount;
    VkDeviceSize vertexStride;
    VkFormat vertexFormat;
    VkBuffer indexData;
    VkDeviceSize indexOffset;
    uint32_t indexCount;
    VkIndexType indexType;
    VkBuffer transformData;
    VkDeviceSize transformOffset;
} VkGeometryTrianglesNV;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **vertexData** is the buffer containing vertex data for this geometry.
- **vertexOffset** is the offset in bytes within **vertexData** containing vertex data for this geometry.
- **vertexCount** is the number of valid vertices.
- **vertexStride** is the stride in bytes between each vertex.
- **vertexFormat** is a **VkFormat** describing the format of each vertex element.
- **indexData** is the buffer containing index data for this geometry.
- **indexOffset** is the offset in bytes within **indexData** containing index data for this geometry.
- **indexCount** is the number of indices to include in this geometry.
- **indexType** is a **VkIndexType** describing the format of each index.
- **transformData** is an optional buffer containing an **VkTransformMatrixNV** structure defining a transformation to be applied to this geometry.
- **transformOffset** is the offset in bytes in **transformData** of the transform information described above.

If **indexType** is **VK_INDEX_TYPE_NONE_NV**, then this structure describes a set of triangles determined by **vertexCount**. Otherwise, this structure describes a set of indexed triangles determined by **indexCount**.
Valid Usage

- **vertexOffset** must be less than the size of **vertexData**
- **vertexOffset** must be a multiple of the component size of **vertexFormat**
- **vertexFormat** must be one of **VK_FORMAT_R32G32B32_SFLOAT**, **VK_FORMAT_R32G32_SFLOAT**, **VK_FORMAT_R16G16B16_SFLOAT**, **VK_FORMAT_R16G16_SFLOAT**, **VK_FORMAT_R16G16_SNORM**, or **VK_FORMAT_R16G16B16_SNORM**
- **vertexStride** must be less than or equal to $2^{32} - 1$
- **indexOffset** must be less than the size of **indexData**
- **indexOffset** must be a multiple of the element size of **indexType**
- **indexType** must be **VK_INDEX_TYPE_UINT16**, **VK_INDEX_TYPE_UINT32**, or **VK_INDEX_TYPE_NONE_NV**
- **indexData** must be **VK_NULL_HANDLE** if **indexType** is **VK_INDEX_TYPE_NONE_NV**
- **indexData** must be a valid **VkBuffer** handle if **indexType** is not **VK_INDEX_TYPE_NONE_NV**
- **indexCount** must be 0 if **indexType** is **VK_INDEX_TYPE_NONE_NV**
- **transformOffset** must be less than the size of **transformData**
- **transformOffset** must be a multiple of 16
Valid Usage (Implicit)

- VUID-VkGeometryTrianglesNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_NV

- VUID-VkGeometryTrianglesNV-pNext-pNext
  pNext must be NULL

- VUID-VkGeometryTrianglesNV-vertexData-parameter
  If vertexData is not VK_NULL_HANDLE, vertexData must be a valid VkBuffer handle

- VUID-VkGeometryTrianglesNV-vertexFormat-parameter
  vertexFormat must be a valid VkFormat value

- VUID-VkGeometryTrianglesNV-indexData-parameter
  If indexData is not VK_NULL_HANDLE, indexData must be a valid VkBuffer handle

- VUID-VkGeometryTrianglesNV-indexType-parameter
  indexType must be a valid VkIndexType value

- VUID-VkGeometryTrianglesNV-transformData-parameter
  If transformData is not VK_NULL_HANDLE, transformData must be a valid VkBuffer handle

- VUID-VkGeometryTrianglesNV-commonparent
  Each of indexData, transformData, and vertexData that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

The VkGeometryAABBNV structure specifies axis-aligned bounding box geometry in a bottom-level acceleration structure, and is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkGeometryAABBNV {
    VkStructureType sType;
    const void* pNext;
    VkBuffer aabbData;
    uint32_t numAABBs;
    uint32_t stride;
    VkDeviceSize offset;
} VkGeometryAABBNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **aabbData** is the buffer containing axis-aligned bounding box data.
- **numAABBs** is the number of AABBs in this geometry.
- **stride** is the stride in bytes between AABBs in aabbData.
- **offset** is the offset in bytes of the first AABB in aabbData.

The AABB data in memory is six 32-bit floats consisting of the minimum x, y, and z values followed by the maximum x, y, and z values.
Valid Usage

- VUID-VkGeometryAABBNV-offset-02439
  offset must be less than the size of aabbData
- VUID-VkGeometryAABBNV-offset-02440
  offset must be a multiple of 8
- VUID-VkGeometryAABBNV-stride-02441
  stride must be a multiple of 8

Valid Usage (Implicit)

- VUID-VkGeometryAABBNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_GEOMETRY_AABB_NV
- VUID-VkGeometryAABBNV-pNext-pNext
  pNext must be NULL
- VUID-VkGeometryAABBNV-aabbData-parameter
  If aabbData is not VK_NULL_HANDLE, aabbData must be a valid VkBuffer handle

To destroy an acceleration structure, call:

```c
// Provided by VK_KHR_acceleration_structure
void vkDestroyAccelerationStructureKHR(
    VkDevice                                    device,
    VkAccelerationStructureKHR                  accelerationStructure,
    const VkAllocationCallbacks*                pAllocator);
```

- device is the logical device that destroys the buffer.
- accelerationStructure is the acceleration structure to destroy.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyAccelerationStructureKHR-accelerationStructure-02442
  All submitted commands that refer to accelerationStructure must have completed execution
- VUID-vkDestroyAccelerationStructureKHR-accelerationStructure-02443
  If VkAllocationCallbacks were provided when accelerationStructure was created, a compatible set of callbacks must be provided here
- VUID-vkDestroyAccelerationStructureKHR-accelerationStructure-02444
  If no VkAllocationCallbacks were provided when accelerationStructure was created, pAllocator must be NULL
Valid Usage (Implicit)

- VUID-vkDestroyAccelerationStructureKHR-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkDestroyAccelerationStructureKHR-accelerationStructure-parameter
  
  If **accelerationStructure** is not **VK_NULL_HANDLE**, **accelerationStructure** must be a valid **VkAccelerationStructureKHR** handle

- VUID-vkDestroyAccelerationStructureKHR-pAllocator-parameter
  
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkDestroyAccelerationStructureKHR-accelerationStructure-parent
  
  If **accelerationStructure** is a valid handle, it must have been created, allocated, or retrieved from **device**

Host Synchronization

- Host access to **accelerationStructure** must be externally synchronized

To destroy an acceleration structure, call:

```c
// Provided by VK_NV_ray_tracing
void vkDestroyAccelerationStructureNV(
    VkDevice device, 
    VkAccelerationStructureNV accelerationStructure, 
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the logical device that destroys the buffer.
- **accelerationStructure** is the acceleration structure to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyAccelerationStructureNV-accelerationStructure-03752
  
  All submitted commands that refer to **accelerationStructure** must have completed execution

- VUID-vkDestroyAccelerationStructureNV-accelerationStructure-03753
  
  If **VkAllocationCallbacks** were provided when **accelerationStructure** was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyAccelerationStructureNV-accelerationStructure-03754
  
  If no **VkAllocationCallbacks** were provided when **accelerationStructure** was created, **pAllocator** must be **NULL**
Valid Usage (Implicit)

- VUID-vkDestroyAccelerationStructureNV-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkDestroyAccelerationStructureNV-accelerationStructure-parameter
  
  If `accelerationStructure` is not `VK_NULL_HANDLE`, `accelerationStructure` must be a valid `VkAccelerationStructureNV` handle

- VUID-vkDestroyAccelerationStructureNV-pAllocator-parameter
  
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkDestroyAccelerationStructureNV-accelerationStructure-parent
  
  If `accelerationStructure` is a valid handle, it must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `accelerationStructure` must be externally synchronized

An acceleration structure has memory requirements for the structure object itself, scratch space for the build, and scratch space for the update.

Scratch space is allocated as a `VkBuffer`, so for `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV` and `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV` the `pMemoryRequirements->alignment` and `pMemoryRequirements->memoryTypeBits` values returned by this call must be filled with zero, and should be ignored by the application.

To query the memory requirements, call:

```c
// Provided by VK_NV_ray_tracing
void vkGetAccelerationStructureMemoryRequirementsNV(
    VkDevice device,                           
    const VkAccelerationStructureMemoryRequirementsInfoNV* pInfo,  
    VkMemoryRequirements2KHR* pMemoryRequirements);
```

- `device` is the logical device on which the acceleration structure was created.
- `pInfo` specifies the acceleration structure to get memory requirements for.
- `pMemoryRequirements` returns the requested acceleration structure memory requirements.
Valid Usage (Implicit)

- **VUID-vkGetAccelerationStructureMemoryRequirementsNV-device-parameter**
  - `device` must be a valid `VkDevice` handle

- **VUID-vkGetAccelerationStructureMemoryRequirementsNV-pInfo-parameter**
  - `pInfo` must be a valid pointer to a valid `VkAccelerationStructureMemoryRequirementsInfoNV` structure

- **VUID-vkGetAccelerationStructureMemoryRequirementsNV-pMemoryRequirements-parameter**
  - `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements2KHR` structure

The `VkAccelerationStructureMemoryRequirementsInfoNV` structure is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkAccelerationStructureMemoryRequirementsInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureMemoryRequirementsTypeNV type;
    VkAccelerationStructureNV accelerationStructure;
} VkAccelerationStructureMemoryRequirementsInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `type` selects the type of memory requirement being queried.
  - `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV` returns the memory requirements for the object itself.
  - `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV` returns the memory requirements for the scratch memory when doing a build.
  - `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV` returns the memory requirements for the scratch memory when doing an update.
- `accelerationStructure` is the acceleration structure to be queried for memory requirements.
Possible values of `type` in `VkAccelerationStructureMemoryRequirementsInfoNV` are:

```c
typedef enum VkAccelerationStructureMemoryRequirementsTypeNV {
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV = 0,
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV = 1,
    VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV = 2,
} VkAccelerationStructureMemoryRequirementsTypeNV;
```

- **VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV** requests the memory requirement for the `VkAccelerationStructureNV` backing store.
- **VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV** requests the memory requirement for scratch space during the initial build.
- **VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV** requests the memory requirement for scratch space during an update.

Possible values of `buildType` in `vkGetAccelerationStructureBuildSizesKHR` are:

```c
typedef enum VkAccelerationStructureBuildTypeKHR {
    VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_KHR = 0,
    VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR = 1,
    VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_OR_DEVICE_KHR = 2,
} VkAccelerationStructureBuildTypeKHR;
```

- **VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_KHR** requests the memory requirement for operations performed by the host.
- **VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR** requests the memory requirement for operations performed by the device.
- **VK_ACCELERATION_STRUCTURE_BUILD_TYPE_HOST_OR_DEVICE_KHR** requests the memory requirement for operations performed by either the host, or the device.

To attach memory to one or more acceleration structures at a time, call:

```c
// Provided by VK_NV_ray_tracing
VkResult vkBindAccelerationStructureMemoryNV(
    VkDevice                                    device,
    uint32_t                                    bindInfoCount,
    const VkBindAccelerationStructureMemoryInfoNV* pBindInfos);
```

- **device** is the logical device that owns the acceleration structures and memory.
- **bindInfoCount** is the number of elements in **pBindInfos**.
- **pBindInfos** is a pointer to an array of `VkBindAccelerationStructureMemoryInfoNV` structures describing acceleration structures and memory to bind.
Valid Usage (Implicit)

- VUID-vkBindAccelerationStructureMemoryNV-device-parameter
  
  `device` must be a valid `VkDevice` handle.

- VUID-vkBindAccelerationStructureMemoryNV-pBindInfos-parameter
  
  `pBindInfos` must be a valid pointer to an array of `bindInfoCount` valid `VkBindAccelerationStructureMemoryInfoNV` structures.

- VUID-vkBindAccelerationStructureMemoryNV-bindInfoCount-arraylength
  
  `bindInfoCount` must be greater than 0.

Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkBindAccelerationStructureMemoryInfoNV` structure is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkBindAccelerationStructureMemoryInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureNV accelerationStructure;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
    uint32_t deviceIndexCount;
    const uint32_t* pDeviceIndices;
} VkBindAccelerationStructureMemoryInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `accelerationStructure` is the acceleration structure to be attached to memory.
- `memory` is a `VkDeviceMemory` object describing the device memory to attach.
- `memoryOffset` is the start offset of the region of memory that is to be bound to the acceleration structure. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified acceleration structure.
- `deviceIndexCount` is the number of elements in `pDeviceIndices`.
- `pDeviceIndices` is a pointer to an array of device indices.
Valid Usage

• VUID-VkBindAccelerationStructureMemoryInfoNV-accelerationStructure-03620
  \texttt{accelerationStructure} must not already be backed by a memory object

• VUID-VkBindAccelerationStructureMemoryInfoNV-memoryOffset-03621
  \texttt{memoryOffset} must be less than the size of \texttt{memory}

• VUID-VkBindAccelerationStructureMemoryInfoNV-memory-03622
  \texttt{memory} must have been allocated using one of the memory types allowed in the
  \texttt{memoryTypeBits} member of the \texttt{VkMemoryRequirements} structure returned from a call to
  \texttt{vkGetAccelerationStructureMemoryRequirementsNV} with \texttt{accelerationStructure} and type
  of \texttt{VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV}

• VUID-VkBindAccelerationStructureMemoryInfoNV-memoryOffset-03623
  \texttt{memoryOffset} must be an integer multiple of the \texttt{alignment} member of the
  \texttt{VkMemoryRequirements} structure returned from a call to
  \texttt{vkGetAccelerationStructureMemoryRequirementsNV} with \texttt{accelerationStructure} and type
  of \texttt{VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV}

• VUID-VkBindAccelerationStructureMemoryInfoNV-size-03624
  The \texttt{size} member of the \texttt{VkMemoryRequirements} structure returned from a call to
  \texttt{vkGetAccelerationStructureMemoryRequirementsNV} with \texttt{accelerationStructure} and type
  of \texttt{VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_OBJECT_NV} must be less than or
  equal to the size of \texttt{memory} minus \texttt{memoryOffset}

Valid Usage (Implicit)

• VUID-VkBindAccelerationStructureMemoryInfoNV-sType-sType
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_BIND_ACCELERATION_STRUCTURE_MEMORY_INFO_NV}

• VUID-VkBindAccelerationStructureMemoryInfoNV-pNext-pNext
  \texttt{pNext} must be \texttt{NULL}

• VUID-VkBindAccelerationStructureMemoryInfoNV-accelerationStructure-parameter
  \texttt{accelerationStructure} must be a valid \texttt{VkAccelerationStructureNV} handle

• VUID-VkBindAccelerationStructureMemoryInfoNV-memory-parameter
  \texttt{memory} must be a valid \texttt{VkDeviceMemory} handle

• VUID-VkBindAccelerationStructureMemoryInfoNV-pDeviceIndices-parameter
  If \texttt{deviceIndexCount} is not 0, \texttt{pDeviceIndices} must be a valid pointer to an array of
  \texttt{deviceIndexCount} \texttt{uint32_t} values

• VUID-VkBindAccelerationStructureMemoryInfoNV-commonparent
  Both of \texttt{accelerationStructure}, and \texttt{memory} must have been created, allocated, or retrieved
  from the same \texttt{VkDevice}

To allow constructing geometry instances with device code if desired, we need to be able to query a
opaque handle for an acceleration structure. This handle is a value of 8 bytes. To get this handle, call:
// Provided by VK_NV_ray_tracing
VkResult vkGetAccelerationStructureHandleNV(
    VkDevice                                    device,
    VkAccelerationStructureNV                   accelerationStructure,
    size_t                                      dataSize,
    void*                                       pData);

• **device** is the logical device that owns the acceleration structures.
• **accelerationStructure** is the acceleration structure.
• **dataSize** is the size in bytes of the buffer pointed to by **pData**.
• **pData** is a pointer to a user-allocated buffer where the results will be written.

### Valid Usage

- VUID-vkGetAccelerationStructureHandleNV-dataSize-02240
  **dataSize** must be large enough to contain the result of the query, as described above
- VUID-vkGetAccelerationStructureHandleNV-accelerationStructure-02787
  **accelerationStructure** must be bound completely and contiguously to a single
  VkDeviceMemory object via **vkBindAccelerationStructureMemoryNV**

### Valid Usage (Implicit)

- VUID-vkGetAccelerationStructureHandleNV-device-parameter
  **device** must be a valid **VkDevice** handle
- VUID-vkGetAccelerationStructureHandleNV-accelerationStructure-parameter
  **accelerationStructure** must be a valid **VkAccelerationStructureNV** handle
- VUID-vkGetAccelerationStructureHandleNV-pData-parameter
  **pData** must be a valid pointer to an array of **dataSize** bytes
- VUID-vkGetAccelerationStructureHandleNV-dataSize-arraylength
  **dataSize** must be greater than 0
- VUID-vkGetAccelerationStructureHandleNV-accelerationStructure-parent
  **accelerationStructure** must have been created, allocated, or retrieved from **device**
Return Codes

Success

• VK_SUCCESS

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

To query the 64-bit device address for an acceleration structure, call:

```c
// Provided by VK_KHR_acceleration_structure
VkDeviceAddress vkGetAccelerationStructureDeviceAddressKHR(
    VkDevice device,
    const VkAccelerationStructureDeviceAddressInfoKHR* pInfo);
```

- `device` is the logical device that the accelerationStructure was created on.
- `pInfo` is a pointer to a `VkAccelerationStructureDeviceAddressInfoKHR` structure specifying the acceleration structure to retrieve an address for.

The 64-bit return value is an address of the acceleration structure, which can be used for device and shader operations that involve acceleration structures, such as ray traversal and acceleration structure building.

If the acceleration structure was created with a non-zero value of `VkAccelerationStructureCreateInfoKHR::deviceAddress` the return value will be the same address.

If the acceleration structure was created with a type of `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR` the returned address must be consistent with the relative offset to other acceleration structures of type of `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR` allocated with the same `VkBuffer`. That is, the difference in returned addresses between the two must be the same as the difference in offsets provided at acceleration structure creation.

**Note**
The acceleration structure device address may be different from the buffer device address corresponding to the acceleration structure’s start offset in its storage buffer for acceleration structure types other than `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR`.

Valid Usage

- VUID-vkGetAccelerationStructureDeviceAddressKHR-device-03504
  
  If `device` was created with multiple physical devices, then the `bufferDeviceAddressMultiDevice` feature must be enabled
Valid Usage (Implicit)

- VUID-vkGetAccelerationStructureDeviceAddressKHR-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetAccelerationStructureDeviceAddressKHR-pInfo-parameter
pInfo must be a valid pointer to a valid VkAccelerationStructureDeviceAddressInfoKHR structure

The VkAccelerationStructureDeviceAddressInfoKHR structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureDeviceAddressInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureKHR accelerationStructure;
} VkAccelerationStructureDeviceAddressInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- accelerationStructure specifies the acceleration structure whose address is being queried.

Valid Usage (Implicit)

- VUID-VkAccelerationStructureDeviceAddressInfoKHR-sType-sType
sType must be VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR

- VUID-VkAccelerationStructureDeviceAddressInfoKHR-pNext-pNext
pNext must be NULL

- VUID-VkAccelerationStructureDeviceAddressInfoKHR-accelerationStructure-parameter
accelerationStructure must be a valid VkAccelerationStructureKHR handle

12.7. Resource Memory Association

Resources are initially created as virtual allocations with no backing memory. Device memory is allocated separately (see Device Memory) and then associated with the resource. This association is done differently for sparse and non-sparse resources.

Resources created with any of the sparse creation flags are considered sparse resources. Resources created without these flags are non-sparse. The details on resource memory association for sparse resources is described in Sparse Resources.

Non-sparse resources must be bound completely and contiguously to a single VkDeviceMemory object before the resource is passed as a parameter to any of the following operations:
• creating image or buffer views
• updating descriptor sets
• recording commands in a command buffer

Once bound, the memory binding is immutable for the lifetime of the resource.

In a logical device representing more than one physical device, buffer and image resources exist on all physical devices but can be bound to memory differently on each. Each such replicated resource is an instance of the resource. For sparse resources, each instance can be bound to memory arbitrarily differently. For non-sparse resources, each instance can either be bound to the local or a peer instance of the memory, or for images can be bound to rectangular regions from the local and/or peer instances. When a resource is used in a descriptor set, each physical device interprets the descriptor according to its own instance’s binding to memory.

**Note**

There are no new copy commands to transfer data between physical devices. Instead, an application can create a resource with a peer mapping and use it as the source or destination of a transfer command executed by a single physical device to copy the data from one physical device to another.

To determine the memory requirements for a buffer resource, call:

```c
// Provided by VK_VERSION_1_0
void vkGetBufferMemoryRequirements(
    device,
    buffer,
    pMemoryRequirements);
```

- `device` is the logical device that owns the buffer.
- `buffer` is the buffer to query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements` structure in which the memory requirements of the buffer object are returned.

**Valid Usage (Implicit)**

- `VUID-vkGetBufferMemoryRequirements-device-parameter`  
  `device` must be a valid `VkDevice` handle

- `VUID-vkGetBufferMemoryRequirements-buffer-parameter`  
  `buffer` must be a valid `VkBuffer` handle

- `VUID-vkGetBufferMemoryRequirements-pMemoryRequirements-parameter`  
  `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements` structure

- `VUID-vkGetBufferMemoryRequirements-buffer-parent`  
  `buffer` must have been created, allocated, or retrieved from `device`
To determine the memory requirements for an image resource which is not created with the
VK_IMAGE_CREATE_DISJOINT_BIT flag set, call:

```c
// Provided by VK_VERSION_1_0
void vkGetImageMemoryRequirements(
    VkDevice                                    device,
    VkImage                                     image,
    VkMemoryRequirements*                       pMemoryRequirements);
```

- **device** is the logical device that owns the image.
- **image** is the image to query.
- **pMemoryRequirements** is a pointer to a `VkMemoryRequirements` structure in which the memory
  requirements of the image object are returned.

### Valid Usage

- **VUID-vkGetImageMemoryRequirements-image-01588**
  - `image` must not have been created with the VK_IMAGE_CREATE_DISJOINT_BIT flag set

- **VUID-vkGetImageMemoryRequirements-image-04004**
  - If `image` was created with the VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID
    external memory handle type, then `image` must be bound to memory

### Valid Usage (Implicit)

- **VUID-vkGetImageMemoryRequirements-device-parameter**
  - `device` must be a valid `VkDevice` handle

- **VUID-vkGetImageMemoryRequirements-image-parameter**
  - `image` must be a valid `VkImage` handle

- **VUID-vkGetImageMemoryRequirements-pMemoryRequirements-parameter**
  - `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements` structure

- **VUID-vkGetImageMemoryRequirements-image-parent**
  - `image` must have been created, allocated, or retrieved from `device`

The `VkMemoryRequirements` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkMemoryRequirements {
    VkDeviceSize    size;
    VkDeviceSize    alignment;
    uint32_t        memoryTypeBits;
} VkMemoryRequirements;
```
• **size** is the size, in bytes, of the memory allocation **required** for the resource.

• **alignment** is the alignment, in bytes, of the offset within the allocation **required** for the resource.

• **memoryTypeBits** is a bitmask and contains one bit set for every supported memory type for the resource. Bit \( i \) is set if and only if the memory type \( i \) in the `VkPhysicalDeviceMemoryProperties` structure for the physical device is supported for the resource.

The precise size of images that will be bound to external Android hardware buffer memory is unknown until the memory has been imported or allocated, so applications **must** not call `vkGetImageMemoryRequirements` or `vkGetImageMemoryRequirements2` with such an `VkImage` before it has been bound to memory. When importing Android hardware buffer memory, the **allocationSize** can be determined by calling `vkGetAndroidHardwareBufferPropertiesANDROID`. When allocating new memory for a `VkImage` that **can** be exported to an Android hardware buffer, the memory's **allocationSize** **must** be zero; the actual size will be determined by the dedicated image’s parameters. After the memory has been allocated, the amount of space allocated from the memory’s heap **can** be obtained by getting the image’s memory requirements or by calling `vkGetAndroidHardwareBufferPropertiesANDROID` with the Android hardware buffer exported from the memory.

When allocating new memory for a `VkBuffer` that **can** be exported to an Android hardware buffer an application **may** still call `vkGetBufferMemoryRequirements` or `vkGetBufferMemoryRequirements2` with `VkBuffer` before it has been bound to memory.

If the resource being queried was created with the `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT`, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT`, or `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT` external memory handle type, the value of **size** has no meaning and **should** be ignored.

The implementation guarantees certain properties about the memory requirements returned by `vkGetBufferMemoryRequirements2`, `vkGetImageMemoryRequirements2`, `vkGetBufferMemoryRequirements` and `vkGetImageMemoryRequirements`:

• The **memoryTypeBits** member always contains at least one bit set.

• If **buffer** is a `VkBuffer` not created with the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bit set, or if **image** is linear image, then the **memoryTypeBits** member always contains at least one bit set corresponding to a `VkMemoryType` with a `propertyFlags` that has both the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` bit and the `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` bit set. In other words, mappable coherent memory **can** always be attached to these objects.

• If **buffer** was created with `VkExternalMemoryBufferCreateInfo::handleTypes` set to 0 or **image** was created with `VkExternalMemoryImageCreateInfo::handleTypes` set to 0, the **memoryTypeBits** member always contains at least one bit set corresponding to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` bit set.

• The **memoryTypeBits** member is identical for all `VkBuffer` objects created with the same value for the **flags** and **usage** members in the `VkBufferCreateInfo` structure and the **handleTypes** member of the `VkExternalMemoryBufferCreateInfo` structure passed to `vkCreateBuffer`. Further, if `usage1` and `usage2` of type `VkBufferUsageFlags` are such that the bits set in `usage2` are a subset of the bits.
set in `usage1`, and they have the same `flags` and `VkExternalMemoryBufferCreateInfo::handleTypes`, then the bits set in `memoryTypeBits` returned for `usage1` must be a subset of the bits set in `memoryTypeBits` returned for `usage2`, for all values of `flags`.

- The `alignment` member is a power of two.
- The `alignment` member is identical for all `VkBuffer` objects created with the same combination of values for the `usage` and `flags` members in the `VkBufferCreateInfo` structure passed to `vkCreateBuffer`.
- The `alignment` member satisfies the buffer descriptor offset alignment requirements associated with the `VkBuffer`'s `usage`:
  - If `usage` included `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT`, `alignment` must be an integer multiple of `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.
  - If `usage` included `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT`, `alignment` must be an integer multiple of `VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment`.
  - If `usage` included `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT`, `alignment` must be an integer multiple of `VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment`.
- For images created with a color format, the `memoryTypeBits` member is identical for all `VkImage` objects created with the same combination of values for the `tiling` member, the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` bit of the `flags` member, the `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` bit of the `flags` member, `handleTypes` member of `VkExternalMemoryImageCreateInfo`, and the `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` of the `usage` member in the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- For images created with a depth/stencil format, the `memoryTypeBits` member is identical for all `VkImage` objects created with the same combination of values for the `format` member, the `tiling` member, the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` bit of the `flags` member, the `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` bit of the `flags` member, `handleTypes` member of `VkExternalMemoryImageCreateInfo`, and the `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` of the `usage` member in the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- If the memory requirements are for a `VkImage`, the `memoryTypeBits` member must not refer to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set if the `image` did not have `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` bit set in the `usage` member of the `VkImageCreateInfo` structure passed to `vkCreateImage`.
- If the memory requirements are for a `VkBuffer`, the `memoryTypeBits` member must not refer to a `VkMemoryType` with a `propertyFlags` that has the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The implication of this requirement is that lazily allocated memory is disallowed for buffers in all cases.</td>
</tr>
</tbody>
</table>

- The `size` member is identical for all `VkBuffer` objects created with the same combination of creation parameters specified in `VkBufferCreateInfo` and its `pNext` chain.
- The `size` member is identical for all `VkImage` objects created with the same combination of creation parameters specified in `VkImageCreateInfo` and its `pNext` chain.
Note
This, however, does not imply that they interpret the contents of the bound memory identically with each other. That additional guarantee, however, can be explicitly requested using `VK_IMAGE_CREATE_ALIAS_BIT`.

To determine the memory requirements for a buffer resource, call:

```c
// Provided by VK_VERSION_1_1
void vkGetBufferMemoryRequirements2(
    VkDevice device,
    const VkBufferMemoryRequirementsInfo2* pInfo,
    VkMemoryRequirements2* pMemoryRequirements);
```

or the equivalent command

```c
// Provided by VK_KHR_get_memory_requirements2
void vkGetBufferMemoryRequirements2KHR(
    VkDevice device,
    const VkBufferMemoryRequirementsInfo2* pInfo,
    VkMemoryRequirements2* pMemoryRequirements);
```

- `device` is the logical device that owns the buffer.
- `pInfo` is a pointer to a `VkBufferMemoryRequirementsInfo2` structure containing parameters required for the memory requirements query.
- `pMemoryRequirements` is a pointer to a `VkMemoryRequirements2` structure in which the memory requirements of the buffer object are returned.

**Valid Usage (Implicit)**

- `VUID-vkGetBufferMemoryRequirements2-device-parameter`  
  `device` must be a valid `VkDevice` handle

- `VUID-vkGetBufferMemoryRequirements2-pInfo-parameter`  
  `pInfo` must be a valid pointer to a valid `VkBufferMemoryRequirementsInfo2` structure

- `VUID-vkGetBufferMemoryRequirements2-pMemoryRequirements-parameter`  
  `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements2` structure

The `VkBufferMemoryRequirementsInfo2` structure is defined as:
```c
// Provided by VK_VERSION_1_1
typedef struct VkBufferMemoryRequirementsInfo2 {
    VkStructureType    sType;
    const void*        pNext;
    VkBuffer           buffer;
} VkBufferMemoryRequirementsInfo2;
```

or the equivalent

```c
// Provided by VK_KHR_get_memory_requirements2
typedef VkBufferMemoryRequirementsInfo2 VkBufferMemoryRequirementsInfo2KHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **buffer** is the buffer to query.

### Valid Usage (Implicit)

- VUID-VkBufferMemoryRequirementsInfo2-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2**

- VUID-VkBufferMemoryRequirementsInfo2-pNext-pNext
  
  **pNext** must be **NULL**

- VUID-VkBufferMemoryRequirementsInfo2-buffer-parameter
  
  **buffer** must be a valid **VkBuffer** handle

To determine the memory requirements for an image resource, call:

```c
// Provided by VK_VERSION_1_1
void vkGetImageMemoryRequirements2(
    VkDevice                                    device,
    const VkImageMemoryRequirementsInfo2*       pInfo,
    VkMemoryRequirements2*                      pMemoryRequirements);
```

or the equivalent command

```c
// Provided by VK_KHR_get_memory_requirements2
void vkGetImageMemoryRequirements2KHR(
    VkDevice                                    device,
    const VkImageMemoryRequirementsInfo2*       pInfo,
    VkMemoryRequirements2*                      pMemoryRequirements);
```

- **device** is the logical device that owns the image.
• pInfo is a pointer to a VkImageMemoryRequirementsInfo2 structure containing parameters required for the memory requirements query.

• pMemoryRequirements is a pointer to a VkMemoryRequirements2 structure in which the memory requirements of the image object are returned.

Valid Usage (Implicit)

• VUID-vkGetImageMemoryRequirements2-device-parameter
device must be a valid VkDevice handle

• VUID-vkGetImageMemoryRequirements2-pInfo-parameter
pInfo must be a valid pointer to a valid VkImageMemoryRequirementsInfo2 structure

• VUID-vkGetImageMemoryRequirements2-pMemoryRequirements-parameter
pMemoryRequirements must be a valid pointer to a VkMemoryRequirements2 structure

The VkImageMemoryRequirementsInfo2 structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageMemoryRequirementsInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
} VkImageMemoryRequirementsInfo2;
```

or the equivalent

```c
// Provided by VK_KHR_get_memory_requirements2
typedef VkImageMemoryRequirementsInfo2 VkImageMemoryRequirementsInfo2KHR;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• image is the image to query.
Valid Usage

- VUID-VkImageMemoryRequirementsInfo2-image-01589
  If image was created with a multi-planar format and the VK_IMAGE_CREATE_DISJOINT_BIT flag, there must be a VkImagePlaneMemoryRequirementsInfo included in the pNext chain of the VkImageMemoryRequirementsInfo2 structure.

- VUID-VkImageMemoryRequirementsInfo2-image-02279
  If image was created with VK_IMAGE_CREATE_DISJOINT_BIT and with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then there must be a VkImagePlaneMemoryRequirementsInfo included in the pNext chain of the VkImageMemoryRequirementsInfo2 structure.

- VUID-VkImageMemoryRequirementsInfo2-image-01590
  If image was not created with the VK_IMAGE_CREATE_DISJOINT_BIT flag, there must not be a VkImagePlaneMemoryRequirementsInfo included in the pNext chain of the VkImageMemoryRequirementsInfo2 structure.

- VUID-VkImageMemoryRequirementsInfo2-image-02280
  If image was created with a single-plane format and with any tiling other than VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then there must not be a VkImagePlaneMemoryRequirementsInfo included in the pNext chain of the VkImageMemoryRequirementsInfo2 structure.

- VUID-VkImageMemoryRequirementsInfo2-image-01897
  If image was created with the VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID external memory handle type, then image must be bound to memory.

Valid Usage (Implicit)

- VUID-VkImageMemoryRequirementsInfo2-sType-sType
  sType must be VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2

- VUID-VkImageMemoryRequirementsInfo2-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkImagePlaneMemoryRequirementsInfo

- VUID-VkImageMemoryRequirementsInfo2-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkImageMemoryRequirementsInfo2-image-parameter
  image must be a valid VkImage handle

To determine the memory requirements for a plane of a disjoint image, add a VkImagePlaneMemoryRequirementsInfo structure to the pNext chain of the VkImageMemoryRequirementsInfo2 structure.

The VkImagePlaneMemoryRequirementsInfo structure is defined as:
or the equivalent

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkImagePlaneMemoryRequirementsInfo VkImagePlaneMemoryRequirementsInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `planeAspect` is the aspect corresponding to the image plane to query.

### Valid Usage

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-02281**
  If the image's tiling is `VK_IMAGE_TILING_LINEAR` or `VK_IMAGE_TILING_OPTIMAL`, then `planeAspect` must be a single valid format plane for the image (that is, for a two-plane image `planeAspect` must be `VK_IMAGE_ASPECT_PLANE_0_BIT` or `VK_IMAGE_ASPECT_PLANE_1_BIT`, and for a three-plane image `planeAspect` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT` or `VK_IMAGE_ASPECT_PLANE_2_BIT`).

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-02282**
  If the image's tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then `planeAspect` must be a single valid memory plane for the image (that is, `aspectMask` must specify a plane index that is less than the `VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount` associated with the image's format and `VkImageDrmFormatModifierPropertiesEXT::drmFormatModifier`).

### Valid Usage (Implicit)

- **VUID-VkImagePlaneMemoryRequirementsInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO`

- **VUID-VkImagePlaneMemoryRequirementsInfo-planeAspect-parameter**
  `planeAspect` must be a valid `VkImageAspectFlagBits` value

The `VkMemoryRequirements2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImagePlaneMemoryRequirementsInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkImageAspectFlagBits    planeAspect;
} VkImagePlaneMemoryRequirementsInfo;
```
or the equivalent

```c
typedef struct VkMemoryRequirements2 {
  VkStructureType sType;
  void* pNext;
  VkMemoryRequirements memoryRequirements;
} VkMemoryRequirements2;
```

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **memoryRequirements** is a **VkMemoryRequirements** structure describing the memory requirements of the resource.

### Valid Usage (Implicit)

- VUID-VkMemoryRequirements2-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2**

- VUID-VkMemoryRequirements2-pNext-pNext

  **pNext** must be **NULL** or a pointer to a valid instance of **VkMemoryDedicatedRequirements**

- VUID-VkMemoryRequirements2-sType-unique

  The **sType** value of each struct in the **pNext** chain must be unique

To determine the dedicated allocation requirements of a buffer or image resource, add a **VkMemoryDedicatedRequirements** structure to the **pNext** chain of the **VkMemoryRequirements2** structure passed as the **pMemoryRequirements** parameter of **vkGetBufferMemoryRequirements2** or **vkGetImageMemoryRequirements2**.

The **VkMemoryDedicatedRequirements** structure is defined as:

```c
typedef struct VkMemoryDedicatedRequirements {
  VkStructureType sType;
  void* pNext;
  VkBool32 prefersDedicatedAllocation;
  VkBool32 requiresDedicatedAllocation;
} VkMemoryDedicatedRequirements;
```

or the equivalent

```c
typedef VkMemoryRequirements2 VkMemoryRequirements2KHR;
```
typedef VkMemoryDedicatedRequirements VkMemoryDedicatedRequirementsKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **prefersDedicatedAllocation** specifies that the implementation would prefer a dedicated allocation for this resource. The application is still free to suballocate the resource but it may get better performance if a dedicated allocation is used.
- **requiresDedicatedAllocation** specifies that a dedicated allocation is required for this resource.

When the implementation sets **requiresDedicatedAllocation** to **VK_TRUE**, it must also set **prefersDedicatedAllocation** to **VK_TRUE**.

If the **VkMemoryDedicatedRequirements** structure is included in the **pNext** chain of the **VkMemoryRequirements2** structure passed as the **pMemoryRequirements** parameter of a **vkGetBufferMemoryRequirements2** call, **requiresDedicatedAllocation** may be **VK_TRUE** under one of the following conditions:

- The **pNext** chain of **VkBufferCreateInfo** for the call to **vkCreateBuffer** used to create the buffer being queried included a **VkExternalMemoryBufferCreateInfo** structure, and any of the handle types specified in **VkExternalMemoryBufferCreateInfo::handleTypes** requires dedicated allocation, as reported by **vkGetPhysicalDeviceExternalBufferProperties** in **VkExternalBufferProperties::externalMemoryProperties.externalMemoryFeatures**, the **requiresDedicatedAllocation** field will be set to **VK_TRUE**.

In all other cases, **requiresDedicatedAllocation** must be set to **VK_FALSE** by the implementation whenever a **VkMemoryDedicatedRequirements** structure is included in the **pNext** chain of the **VkMemoryRequirements2** structure passed to a call to **vkGetBufferMemoryRequirements2**.

If the **VkMemoryDedicatedRequirements** structure is included in the **pNext** chain of the **VkMemoryRequirements2** structure passed as the **pMemoryRequirements** parameter of a **vkGetBufferMemoryRequirements2** call and **VK_BUFFER_CREATE_SPARSE_BINDING_BIT** was set in **VkBufferCreateInfo::flags** when buffer was created then the implementation must set both **prefersDedicatedAllocation** and **requiresDedicatedAllocation** to **VK_FALSE**.

If the **VkMemoryDedicatedRequirements** structure is included in the **pNext** chain of the **VkMemoryRequirements2** structure passed as the **pMemoryRequirements** parameter of a **vkGetImageMemoryRequirements2** call, **requiresDedicatedAllocation** may be **VK_TRUE** under one of the following conditions:

- The **pNext** chain of **VkImageCreateInfo** for the call to **vkCreateImage** used to create the image being queried included a **VkExternalMemoryImageCreateInfo** structure, and any of the handle types specified in **VkExternalMemoryImageCreateInfo::handleTypes** requires dedicated allocation, as reported by **vkGetPhysicalDeviceImageFormatProperties2** in **VkExternalImageFormatProperties::externalMemoryProperties.externalMemoryFeatures**, the **requiresDedicatedAllocation** field will be set to **VK_TRUE**.
In all other cases, `requiresDedicatedAllocation` must be set to `VK_FALSE` by the implementation whenever a `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed to a call to `vkGetImageMemoryRequirements2`.

If the `VkMemoryDedicatedRequirements` structure is included in the `pNext` chain of the `VkMemoryRequirements2` structure passed as the `pMemoryRequirements` parameter of a `vkGetImageMemoryRequirements2` call and `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` was set in `VkImageCreateInfo::flags` when image was created then the implementation must set both `prefersDedicatedAllocation` and `requiresDedicatedAllocation` to `VK_FALSE`.

Valid Usage (Implicit)

- `VUID-VkMemoryDedicatedRequirements-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS`

To attach memory to a buffer object, call:

```
// Provided by VK_VERSION_1_0
VkResult vkBindBufferMemory(
    VkDevice device,         // Logical device that owns the buffer and memory.
    VkBuffer buffer,        // Buffer to be attached to memory.
    VkDeviceMemory memory,  // Memory to attach, described by a VkDeviceMemory.
    VkDeviceSize memoryOffset);  // Start offset of the memory region to bind to the buffer.
```

- `device` is the logical device that owns the buffer and memory.
- `buffer` is the buffer to be attached to memory.
- `memory` is a `VkDeviceMemory` object describing the device memory to attach.
- `memoryOffset` is the start offset of the region of `memory` which is to be bound to the buffer. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified buffer.

`vkBindBufferMemory` is equivalent to passing the same parameters through `VkBindBufferMemoryInfo` to `vkBindBufferMemory2`. 
Valid Usage

- **VUID-vkBindBufferMemory-buffer-01029**
  
  *buffer must not already be backed by a memory object*

- **VUID-vkBindBufferMemory-buffer-01030**
  
  *buffer must not have been created with any sparse memory binding flags*

- **VUID-vkBindBufferMemory-memoryOffset-01031**
  
  *memoryOffset must be less than the size of memory*

- **VUID-vkBindBufferMemory-memory-01035**
  
  *memory must have been allocated using one of the memory types allowed in the memoryTypeBits member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer*

- **VUID-vkBindBufferMemory-memoryOffset-01036**
  
  *memoryOffset must be an integer multiple of the alignment member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer*

- **VUID-vkBindBufferMemory-size-01037**
  
  *The size member of the VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer must be less than or equal to the size of memory minus memoryOffset*

- **VUID-vkBindBufferMemory-buffer-01444**
  
  *If buffer requires a dedicated allocation (as reported by vkGetBufferMemoryRequirements2 in VkMemoryDedicatedRequirements::requiresDedicatedAllocation for buffer), memory must have been created with VkDedicatedAllocationAllocateInfo::buffer equal to buffer*

- **VUID-vkBindBufferMemory-memory-01508**
  
  *If the VkMemoryAllocateInfo provided when memory was allocated included a VkMemoryDedicatedAllocateInfo structure in its pNext chain, and VkMemoryDedicatedAllocateInfo::buffer was not VK_NULL_HANDLE, then buffer must equal VkMemoryDedicatedAllocateInfo::buffer, and memoryOffset must be zero*

- **VUID-vkBindBufferMemory-None-01898**
  
  *If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit set, the buffer must be bound to a memory object allocated with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT*

- **VUID-vkBindBufferMemory-None-01899**
  
  *If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit not set, the buffer must not be bound to a memory object created with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT*

- **VUID-vkBindBufferMemory-buffer-01038**
  
  *If buffer was created with VkDedicatedAllocationBufferCreateInfoNV::dedicatedAllocation equal to VK_TRUE, memory must have been created with VkDedicatedAllocationMemoryAllocateInfoNV::buffer equal to a buffer handle created with identical creation parameters to buffer and memoryOffset must be zero*
If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not 0, it must include at least one of the handles set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

If `memory` was created by a memory import operation, that is not `VkImportAndroidHardwareBufferInfoANDROID` with a non-NULL `buffer` value, the external handle type of the imported memory must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

If `memory` was created with the `VkImportAndroidHardwareBufferInfoANDROID` memory import operation with a non-NULL `buffer` value, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

If the `VkPhysicalDeviceBufferDeviceAddressFeatures::bufferDeviceAddress` feature is enabled and `buffer` was created with the `VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT` bit set, `memory` must have been allocated with the `VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT` bit set.

Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- **buffer** must be a valid `VkBuffer` handle
- **memory** must be a valid `VkDeviceMemory` handle
- **buffer** must have been created, allocated, or retrieved from **device**
- **memory** must have been created, allocated, or retrieved from **device**

Host Synchronization

- Host access to **buffer** must be externally synchronized
### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR`

To attach memory to buffer objects for one or more buffers at a time, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkBindBufferMemory2(
    VkDevice device,  // Logical device that owns the buffers and memory.
    uint32_t bindInfoCount, // Number of elements in pBindInfos.
    const VkBindBufferMemoryInfo* pBindInfos); // Pointer to array of bindInfoCount VkBindBufferMemoryInfo structures describing buffers and memory to bind.
```

or the equivalent command

```c
// Provided by VK_KHR_bind_memory2
VkResult vkBindBufferMemory2KHR(
    VkDevice device,  // Logical device that owns the buffers and memory.
    uint32_t bindInfoCount, // Number of elements in pBindInfos.
    const VkBindBufferMemoryInfo* pBindInfos); // Pointer to array of bindInfoCount VkBindBufferMemoryInfo structures describing buffers and memory to bind.
```

- `device` is the logical device that owns the buffers and memory.
- `bindInfoCount` is the number of elements in `pBindInfos`.
- `pBindInfos` is a pointer to an array of `VkBindBufferMemoryInfo` structures describing buffers and memory to bind.

On some implementations, it **may** be more efficient to batch memory bindings into a single command.
Valid Usage (Implicit)

- **VUID-vkBindBufferMemory2-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkBindBufferMemory2-pBindInfos-parameter**
  
  `pBindInfos` must be a valid pointer to an array of `bindInfoCount` valid `VkBindBufferMemoryInfo` structures

- **VUID-vkBindBufferMemory2-bindInfoCount-arraylength**
  
  `bindInfoCount` must be greater than 0

Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
- **VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR**

`VkBindBufferMemoryInfo` contains members corresponding to the parameters of `vkBindBufferMemory`.

The `VkBindBufferMemoryInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindBufferMemoryInfo {
    VkStructureType    sType;
    const void*        pNext;
    VkBuffer           buffer;
    VkDeviceMemory     memory;
    VkDeviceSize       memoryOffset;
} VkBindBufferMemoryInfo;
```

or the equivalent

```c
// Provided by VK_KHR_bind_memory2
typedef VkBindBufferMemoryInfo VkBindBufferMemoryInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `buffer` is the buffer to be attached to memory.
• **memory** is a *VkDeviceMemory* object describing the device memory to attach.

• **memoryOffset** is the start offset of the region of **memory** which is to be bound to the buffer. The number of bytes returned in the *VkMemoryRequirements::size* member in **memory**, starting from **memoryOffset** bytes, will be bound to the specified buffer.
Valid Usage

- VUID-VkBindBufferMemoryInfo-buffer-01029
  buffer must not already be backed by a memory object

- VUID-VkBindBufferMemoryInfo-buffer-01030
  buffer must not have been created with any sparse memory binding flags

- VUID-VkBindBufferMemoryInfo-memoryOffset-01031
  memoryOffset must be less than the size of memory

- VUID-VkBindBufferMemoryInfo-memory-01035
  memory must have been allocated using one of the memory types allowed in the
  memoryTypeBits member of the VkMemoryRequirements structure returned from a call to
  vkGetBufferMemoryRequirements with buffer

- VUID-VkBindBufferMemoryInfo-memoryOffset-01036
  memoryOffset must be an integer multiple of the alignment member of the
  VkMemoryRequirements structure returned from a call to vkGetBufferMemoryRequirements with buffer

- VUID-VkBindBufferMemoryInfo-size-01037
  The size member of the VkMemoryRequirements structure returned from a call to
  vkGetBufferMemoryRequirements with buffer must be less than or equal to the size of memory
  minus memoryOffset

- VUID-VkBindBufferMemoryInfo-buffer-01444
  If buffer requires a dedicated allocation (as reported by
  vkGetBufferMemoryRequirements2 in VkMemoryDedicatedRequirements
  ::requiresDedicatedAllocation for buffer), memory must have been created with
  VkMemoryDedicatedAllocateInfo::buffer equal to buffer

- VUID-VkBindBufferMemoryInfo-memory-01508
  If the VkMemoryAllocateInfo provided when memory was allocated included a
  VkMemoryDedicatedAllocateInfo structure in its pNext chain, and
  VkMemoryDedicatedAllocateInfo::buffer was not VK_NULL_HANDLE, then buffer must
equal VkMemoryDedicatedAllocateInfo::buffer, and memoryOffset must be zero

- VUID-VkBindBufferMemoryInfo-None-01898
  If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit set, the buffer must
  be bound to a memory object allocated with a memory type that reports
  VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-VkBindBufferMemoryInfo-None-01899
  If buffer was created with the VK_BUFFER_CREATE_PROTECTED_BIT bit not set, the buffer must
  not be bound to a memory object created with a memory type that reports
  VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-VkBindBufferMemoryInfo-buffer-01038
  If buffer was created with VkDedicatedAllocationBufferCreateInfoNV
  ::dedicatedAllocation equal to VK_TRUE, memory must have been created with
  VkDedicatedAllocationMemoryAllocateInfoNV::buffer equal to a buffer handle created
  with identical creation parameters to buffer and memoryOffset must be zero
If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate `memory` is not 0, it must include at least one of the handles set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

If `memory` was created by a memory import operation, that is not `VkImportAndroidHardwareBufferInfoANDROID` with a non-NULL `buffer` value, the external handle type of the imported memory must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

If `memory` was created with the `VkImportAndroidHardwareBufferInfoANDROID` memory import operation with a non-NULL `buffer` value, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` when `buffer` was created.

If the `pNext` chain includes a `VkBindBufferMemoryDeviceGroupInfo` structure, all instances of `memory` specified by `VkBindBufferMemoryDeviceGroupInfo::pDeviceIndices` must have been allocated.

### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO`
- `pNext` must be `NULL` or a pointer to a valid instance of `VkBindBufferMemoryDeviceGroupInfo`
- The `sType` value of each struct in the `pNext` chain must be unique
- `buffer` must be a valid `VkBuffer` handle
- `memory` must be a valid `VkDeviceMemory` handle
- Both of `buffer`, and `memory` must have been created, allocated, or retrieved from the same `VkDevice`
**Valid Usage**

- VUID-VkBindBufferMemoryDeviceGroupInfo-deviceIndexCount-01606
  - `deviceIndexCount` must either be zero or equal to the number of physical devices in the logical device

- VUID-VkBindBufferMemoryDeviceGroupInfo-pDeviceIndices-01607
  - All elements of `pDeviceIndices` must be valid device indices
Valid Usage (Implicit)

- VUID-VkBindBufferMemoryDeviceGroupInfo-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO`

- VUID-VkBindBufferMemoryDeviceGroupInfo-pDeviceIndices-parameter
  
  If `deviceIndexCount` is not 0, `pDeviceIndices` **must** be a valid pointer to an array of `deviceIndexCount` `uint32_t` values

To attach memory to a `VkImage` object created without the `VK_IMAGE_CREATE_DISJOINT_BIT` set, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkBindImageMemory(
    VkDevice                                    device, 
    VkImage                                     image, 
    VkDeviceMemory                              memory, 
    VkDeviceSize                                memoryOffset);
```

- `device` is the logical device that owns the image and memory.
- `image` is the image.
- `memory` is the `VkDeviceMemory` object describing the device memory to attach.
- `memoryOffset` is the start offset of the region of `memory` which is to be bound to the image. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified image.

`vkBindImageMemory` is equivalent to passing the same parameters through `VkBindImageMemoryInfo` to `vkBindImageMemory2`. 
Valid Usage

- VUID-vkBindImageMemory-image-01044
  image must not already be backed by a memory object

- VUID-vkBindImageMemory-image-01045
  image must not have been created with any sparse memory binding flags

- VUID-vkBindImageMemory-memoryOffset-01046
  memoryOffset must be less than the size of memory

- VUID-vkBindImageMemory-image-01445
  If image requires a dedicated allocation (as reported by 
vkGetImageMemoryRequirements2 in VkMemoryDedicatedRequirements
  ::requiresDedicatedAllocation for image), memory must have been created with
  VkMemoryDedicatedAllocateInfo::image equal to image

- VUID-vkBindImageMemory-memory-02628
  If the dedicated allocation image aliasing feature is not enabled, and the
  VkMemoryAllocateInfo provided when memory was allocated included a
  VkMemoryDedicatedAllocateInfo structure in its pNext chain, and
  VkMemoryDedicatedAllocateInfo::image was not VK_NULL_HANDLE, then image must
  equal VkMemoryDedicatedAllocateInfo::image and memoryOffset must be zero

- VUID-vkBindImageMemory-memory-02629
  If the dedicated allocation image aliasing feature is enabled, and the
  VkMemoryAllocateInfo provided when memory was allocated included a
  VkMemoryDedicatedAllocateInfo structure in its pNext chain, and
  VkMemoryDedicatedAllocateInfo::image was not VK_NULL_HANDLE, then memoryOffset must be zero, and image must be either equal to
  VkMemoryDedicatedAllocateInfo::image or an image that was created using the same
  parameters in VkImageCreateInfo, with the exception that extent and arrayLayers may
  differ subject to the following restrictions: every dimension in the extent parameter of the
  image being bound must be equal to or smaller than the original image for which the
  allocation was created; and the arrayLayers parameter of the image being bound must
  be equal to or smaller than the original image for which the allocation was created

- VUID-vkBindImageMemory-None-01901
  If image was created with the VK_IMAGE_CREATE_PROTECTED_BIT bit set, the image must be
  bound to a memory object allocated with a memory type that reports
  VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-vkBindImageMemory-None-01902
  If image was created with the VK_IMAGE_CREATE_PROTECTED_BIT bit not set, the image must not be bound to a memory object created with a memory type that reports
  VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-vkBindImageMemory-image-01050
  If image was created with VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation
equal to VK_TRUE, memory must have been created with
  VkDedicatedAllocationMemoryAllocateInfoNV::image equal to an image handle created
  with identical creation parameters to image and memoryOffset must be zero

- VUID-vkBindImageMemory-memory-02728

Chapter 12. Resource Creation | 863
If the value of `VkExportMemoryAllocateInfo::handleTypes` used to allocate memory is not 0, it must include at least one of the handles set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created.

- **VUID-vkBindImageMemory-memory-02989**
  If memory was created by a memory import operation, that is not `VkImportAndroidHardwareBufferInfoANDROID` with a non-NULL buffer value, the external handle type of the imported memory must also have been set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created.

- **VUID-vkBindImageMemory-memory-02990**
  If memory was created with the `VkImportAndroidHardwareBufferInfoANDROID` memory import operation with a non-NULL buffer value, `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID` must also have been set in `VkExternalMemoryImageCreateInfo::handleTypes` when `image` was created.

- **VUID-vkBindImageMemory-image-01608**
  `image` must not have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` set.

- **VUID-vkBindImageMemory-memory-01047**
  memory must have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with `image`.

- **VUID-vkBindImageMemory-memoryOffset-01048**
  `memoryOffset` must be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with `image`.

- **VUID-vkBindImageMemory-size-01049**
  The difference of the size of `memory` and `memoryOffset` must be greater than or equal to the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements` with the same `image`.

### Valid Usage (Implicit)

- **VUID-vkBindImageMemory-device-parameter**
  `device` must be a valid `VkDevice` handle.

- **VUID-vkBindImageMemory-image-parameter**
  `image` must be a valid `VkImage` handle.

- **VUID-vkBindImageMemory-memory-parameter**
  `memory` must be a valid `VkDeviceMemory` handle.

- **VUID-vkBindImageMemory-image-parent**
  `image` must have been created, allocated, or retrieved from `device`.

- **VUID-vkBindImageMemory-memory-parent**
  `memory` must have been created, allocated, or retrieved from `device`.
Host Synchronization

- Host access to image must be externally synchronized

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To attach memory to image objects for one or more images at a time, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkBindImageMemory2(
    VkDevice device,
    uint32_t bindInfoCount,
    const VkBindImageMemoryInfo* pBindInfos);
```

or the equivalent command

```c
// Provided by VK_KHR_bind_memory2
VkResult vkBindImageMemory2KHR(
    VkDevice device,
    uint32_t bindInfoCount,
    const VkBindImageMemoryInfo* pBindInfos);
```

- `device` is the logical device that owns the images and memory.
- `bindInfoCount` is the number of elements in `pBindInfos`.
- `pBindInfos` is a pointer to an array of `VkBindImageMemoryInfo` structures, describing images and memory to bind.

On some implementations, it may be more efficient to batch memory bindings into a single command.
Valid Usage

- VUID-vkBindImageMemory2-pBindInfos-02858
  If any VkBindImageMemoryInfo::image was created with VK_IMAGE_CREATE_DISJOINT_BIT then all planes of VkBindImageMemoryInfo::image must be bound individually in separate pBindInfos

- VUID-vkBindImageMemory2-pBindInfos-04006
  pBindInfos must not refer to the same image subresource more than once

Valid Usage (Implicit)

- VUID-vkBindImageMemory2-device-parameter
  device must be a valid VkDevice handle

- VUID-vkBindImageMemory2-pBindInfos-parameter
  pBindInfos must be a valid pointer to an array of bindInfoCount valid VkBindImageMemoryInfo structures

- VUID-vkBindImageMemory2-bindInfoCount-arraylength
  bindInfoCount must be greater than 0

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

VkBindImageMemoryInfo contains members corresponding to the parameters of vkBindImageMemory.

The VkBindImageMemoryInfo structure is defined as:

```
// Provided by VK_VERSION_1_1
typedef struct VkBindImageMemoryInfo {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
} VkBindImageMemoryInfo;
```
or the equivalent

```c
// Provided by VK_KHR_bind_memory2
typedef VkBindImageMemoryInfo VkBindImageMemoryInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **image** is the image to be attached to memory.
- **memory** is a `VkDeviceMemory` object describing the device memory to attach.
- **memoryOffset** is the start offset of the region of `memory` which is to be bound to the image. The number of bytes returned in the `VkMemoryRequirements::size` member in `memory`, starting from `memoryOffset` bytes, will be bound to the specified image.
Valid Usage

- VUID-VkBindImageMemoryInfo-image-01044
  image must not already be backed by a memory object

- VUID-VkBindImageMemoryInfo-image-01045
  image must not have been created with any sparse memory binding flags

- VUID-VkBindImageMemoryInfo-memoryOffset-01046
  memoryOffset must be less than the size of memory

- VUID-VkBindImageMemoryInfo-image-01445
  If image requires a dedicated allocation (as reported by vkGetImageMemoryRequirements2 in VkMemoryDedicatedRequirements::requiresDedicatedAllocation for image), memory must have been created with VkMemoryDedicatedAllocateInfo::image equal to image

- VUID-VkBindImageMemoryInfo-memory-02628
  If the dedicated allocation image aliasing feature is not enabled, and the VkMemoryAllocateInfo provided when memory was allocated included a VkMemoryDedicatedAllocateInfo structure in its pNext chain, and VkMemoryDedicatedAllocateInfo::image was not VK_NULL_HANDLE, then image must equal VkMemoryDedicatedAllocateInfo::image and memoryOffset must be zero

- VUID-VkBindImageMemoryInfo-memory-02629
  If the dedicated allocation image aliasing feature is enabled, and the VkMemoryAllocateInfo provided when memory was allocated included a VkMemoryDedicatedAllocateInfo structure in its pNext chain, and VkMemoryDedicatedAllocateInfo::image was not VK_NULL_HANDLE, then memoryOffset must be zero, and image must be either equal to VkMemoryDedicatedAllocateInfo::image or an image that was created using the same parameters in VkImageCreateInfo, with the exception that extent and arrayLayers may differ subject to the following restrictions: every dimension in the extent parameter of the image being bound must be equal to or smaller than the original image for which the allocation was created; and the arrayLayers parameter of the image being bound must be equal to or smaller than the original image for which the allocation was created

- VUID-VkBindImageMemoryInfo-None-01901
  If image was created with the VK_IMAGE_CREATE_PROTECTED_BIT bit set, the image must be bound to a memory object allocated with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-VkBindImageMemoryInfo-None-01902
  If image was created with the VK_IMAGE_CREATE_PROTECTED_BIT bit not set, the image must not be bound to a memory object created with a memory type that reports VK_MEMORY_PROPERTY_PROTECTED_BIT

- VUID-VkBindImageMemoryInfo-image-01050
  If image was created with VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation equal to VK_TRUE, memory must have been created with VkDedicatedAllocationMemoryAllocateInfoNV::image equal to an image handle created with identical creation parameters to image and memoryOffset must be zero

- VUID-VkBindImageMemoryInfo-memory-02728
If the value of \texttt{VkExportMemoryAllocateInfo::handleTypes} used to allocate \texttt{memory} is not 0, it \textbf{must} include at least one of the handles set in \texttt{VkExternalMemoryImageCreateInfo::handleTypes} when \texttt{image} was created.

- \textbf{VUID-VkBindImageMemoryInfo-memory-02989} If \texttt{memory} was created by a memory import operation, that is not \texttt{VkImportAndroidHardwareBufferInfoANDROID} with a non-NULL \texttt{buffer} value, the external handle type of the imported memory \textbf{must} also have been set in \texttt{VkExternalMemoryImageCreateInfo::handleTypes} when \texttt{image} was created.

- \textbf{VUID-VkBindImageMemoryInfo-memory-02990} If \texttt{memory} was created with the \texttt{VkImportAndroidHardwareBufferInfoANDROID} memory import operation with a non-NULL \texttt{buffer} value, \texttt{VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID} \textbf{must} also have been set in \texttt{VkExternalMemoryImageCreateInfo::handleTypes} when \texttt{image} was created.

- \textbf{VUID-VkBindImageMemoryInfo-pNext-01615} If the \texttt{pNext} chain does not include a \texttt{VkBindImagePlaneMemoryInfo} structure, \texttt{memory} \textbf{must} have been allocated using one of the memory types allowed in the \texttt{memoryTypeBits} member of the \texttt{VkMemoryRequirements} structure returned from a call to \texttt{vkGetImageMemoryRequirements2} with \texttt{image}.

- \textbf{VUID-VkBindImageMemoryInfo-pNext-01616} If the \texttt{pNext} chain does not include a \texttt{VkBindImagePlaneMemoryInfo} structure, \texttt{memoryOffset} \textbf{must} be an integer multiple of the \texttt{alignment} member of the \texttt{VkMemoryRequirements} structure returned from a call to \texttt{vkGetImageMemoryRequirements2} with the same \texttt{image}.

- \textbf{VUID-VkBindImageMemoryInfo-pNext-01617} If the \texttt{pNext} chain does not include a \texttt{VkBindImagePlaneMemoryInfo} structure, the difference of the size of \texttt{memory} and \texttt{memoryOffset} \textbf{must} be greater than or equal to the \texttt{size} member of the \texttt{VkMemoryRequirements} structure returned from a call to \texttt{vkGetImageMemoryRequirements2} with the same \texttt{image}.

- \textbf{VUID-VkBindImageMemoryInfo-pNext-01618} If the \texttt{pNext} chain includes a \texttt{VkBindImagePlaneMemoryInfo} structure, \texttt{image} \textbf{must} have been created with the \texttt{VK_IMAGE_CREATE_DISJOINT_BIT} bit set.

- \textbf{VUID-VkBindImageMemoryInfo-pNext-01619} If the \texttt{pNext} chain includes a \texttt{VkBindImagePlaneMemoryInfo} structure, \texttt{memory} \textbf{must} have been allocated using one of the memory types allowed in the \texttt{memoryTypeBits} member of the \texttt{VkMemoryRequirements} structure returned from a call to \texttt{vkGetImageMemoryRequirements2} with \texttt{image} and where \texttt{VkBindImagePlaneMemoryInfo::planeAspect} corresponds to the \texttt{VkImagePlaneMemoryRequirementsInfo::planeAspect} in the \texttt{VkImageMemoryRequirementsInfo2} structure's \texttt{pNext} chain.

- \textbf{VUID-VkBindImageMemoryInfo-pNext-01620} If the \texttt{pNext} chain includes a \texttt{VkBindImagePlaneMemoryInfo} structure, \texttt{memoryOffset} \textbf{must} be an integer multiple of the \texttt{alignment} member of the \texttt{VkMemoryRequirements} structure returned from a call to \texttt{vkGetImageMemoryRequirements2} with \texttt{image} and where \texttt{VkBindImagePlaneMemoryInfo::planeAspect} corresponds to the \texttt{VkImagePlaneMemoryRequirementsInfo::planeAspect} in the
If the `pNext` chain includes a `VkBindImagePlaneMemoryInfo` structure, the difference of the size of `memory` and `memoryOffset` must be greater than or equal to the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetImageMemoryRequirements2` with the same `image` and where `VkBindImagePlaneMemoryInfo::planeAspect` corresponds to the `VkImagePlaneMemoryRequirementsInfo::planeAspect` in the `VkImageMemoryRequirementsInfo2` structure's `pNext` chain.

If the `pNext` chain includes a `VkBindImageMemoryDeviceGroupInfo` structure, all instances of `memory` specified by `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` must have been allocated.

If the `pNext` chain includes a `VkBindImageMemoryDeviceGroupInfo` structure, and `VkBindImageMemoryDeviceGroupInfo::splitInstanceBindRegionCount` is not zero, then `image` must have been created with the `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT` bit set.

If the `pNext` chain includes a `VkBindImageMemoryDeviceGroupInfo` structure, all elements of `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions` must be valid rectangles contained within the dimensions of `image`.

If the `pNext` chain includes a `VkBindImageMemoryDeviceGroupInfo` structure, the union of the areas of all elements of `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions` that correspond to the same instance of `image` must cover the entire image.

If `image` was created with a valid swapchain handle in `VkImageSwapchainCreateInfoKHR::swapchain`, then the `pNext` chain must include a `VkBindImageMemorySwapchainInfoKHR` structure containing the same swapchain handle.

If the `pNext` chain includes a `VkBindImageMemorySwapchainInfoKHR` structure, `memory` must be `VK_NULL_HANDLE`.

If the `pNext` chain does not include a `VkBindImageMemorySwapchainInfoKHR` structure, `memory` must be a valid `VkDeviceMemory` handle.
Valid Usage (Implicit)

- **VUID-VkBindImageMemoryInfo-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO`

- **VUID-VkBindImageMemoryInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkBindImageMemoryDeviceGroupInfo`, `VkBindImageMemorySwapchainInfoKHR`, or `VkBindImagePlaneMemoryInfo`

- **VUID-VkBindImageMemoryInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkBindImageMemoryInfo-image-parameter**
  
  `image` must be a valid `VkImage` handle

- **VUID-VkBindImageMemoryInfo-commonparent**
  
  Both of `image`, and `memory` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

// Provided by VK_VERSION_1_1

typedef struct VkBindImageMemoryDeviceGroupInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           deviceIndexCount;
    const uint32_t*    pDeviceIndices;
    uint32_t           splitInstanceBindRegionCount;
    const VkRect2D*    pSplitInstanceBindRegions;
} VkBindImageMemoryDeviceGroupInfo;

or the equivalent

// Provided by VK_KHR_device_group with VK_KHR_bind_memory2

typedef VkBindImageMemoryDeviceGroupInfo VkBindImageMemoryDeviceGroupInfoKHR;

If the `pNext` list of `VkBindImageMemoryInfo` includes a `VkBindImageMemoryDeviceGroupInfo` structure, then that structure determines how memory is bound to images across multiple devices in a device group.

The `VkBindImageMemoryDeviceGroupInfo` structure is defined as:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **deviceIndexCount** is the number of elements in `pDeviceIndices`.
- **pDeviceIndices** is a pointer to an array of device indices.
- **splitInstanceBindRegionCount** is the number of elements in `pSplitInstanceBindRegions`. 
• pSplitInstanceBindRegions is a pointer to an array of VkRect2D structures describing which regions of the image are attached to each instance of memory.

If deviceIndexCount is greater than zero, then on device index i image is attached to the instance of the memory on the physical device with device index pDeviceIndices[i].

Let N be the number of physical devices in the logical device. If splitInstanceBindRegionCount is greater than zero, then pSplitInstanceBindRegions is an array of $N^2$ rectangles, where the image region specified by the rectangle at element $i*N+j$ in resource instance i is bound to the memory instance j. The blocks of the memory that are bound to each sparse image block region use an offset in memory, relative to memoryOffset, computed as if the whole image were being bound to a contiguous range of memory. In other words, horizontally adjacent image blocks use consecutive blocks of memory, vertically adjacent image blocks are separated by the number of bytes per block multiplied by the width in blocks of image, and the block at (0,0) corresponds to memory starting at memoryOffset.

If splitInstanceBindRegionCount and deviceIndexCount are zero and the memory comes from a memory heap with the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT bit set, then it is as if pDeviceIndices contains consecutive indices from zero to the number of physical devices in the logical device, minus one. In other words, by default each physical device attaches to its own instance of the memory.

If splitInstanceBindRegionCount and deviceIndexCount are zero and the memory comes from a memory heap without the VK_MEMORY_HEAP_MULTI_INSTANCE_BIT bit set, then it is as if pDeviceIndices contains an array of zeros. In other words, by default each physical device attaches to instance zero.
Valid Usage

- VUID-VkBindImageMemoryDeviceGroupInfo-deviceIndexCount-01633
  At least one of `deviceIndexCount` and `splitInstanceBindRegionCount` **must** be zero.

- VUID-VkBindImageMemoryDeviceGroupInfo-deviceIndexCount-01634
  `deviceIndexCount` **must** either be zero or equal to the number of physical devices in the logical device.

- VUID-VkBindImageMemoryDeviceGroupInfo-pDeviceIndices-01635
  All elements of `pDeviceIndices` **must** be valid device indices.

- VUID-VkBindImageMemoryDeviceGroupInfo-splitInstanceBindRegionCount-01636
  `splitInstanceBindRegionCount` **must** either be zero or equal to the number of physical devices in the logical device squared.

- VUID-VkBindImageMemoryDeviceGroupInfo-pSplitInstanceBindRegions-01637
  Elements of `pSplitInstanceBindRegions` that correspond to the same instance of an image **must** not overlap.

- VUID-VkBindImageMemoryDeviceGroupInfo-offset-01638
  The `offset.x` member of any element of `pSplitInstanceBindRegions` **must** be a multiple of the sparse image block width (`VkSparseImageFormatProperties::imageGranularity.width`) of all non-metadata aspects of the image.

- VUID-VkBindImageMemoryDeviceGroupInfo-offset-01639
  The `offset.y` member of any element of `pSplitInstanceBindRegions` **must** be a multiple of the sparse image block height (`VkSparseImageFormatProperties::imageGranularity.height`) of all non-metadata aspects of the image.

- VUID-VkBindImageMemoryDeviceGroupInfo-extent-01640
  The `extent.width` member of any element of `pSplitInstanceBindRegions` **must** either be a multiple of the sparse image block width of all non-metadata aspects of the image, or else `extent.width + offset.x` **must** equal the width of the image subresource.

- VUID-VkBindImageMemoryDeviceGroupInfo-extent-01641
  The `extent.height` member of any element of `pSplitInstanceBindRegions` **must** either be a multiple of the sparse image block height of all non-metadata aspects of the image, or else `extent.height + offset.y` **must** equal the width of the image subresource.
Valid Usage (Implicit)

- VUID-VkBindImageMemoryDeviceGroupInfo-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO

- VUID-VkBindImageMemoryDeviceGroupInfo-pDeviceIndices-parameter
  
  If deviceIndexCount is not 0, pDeviceIndices must be a valid pointer to an array of deviceIndexCount uint32_t values

- VUID-VkBindImageMemoryDeviceGroupInfo-pSplitInstanceBindRegions-parameter
  
  If splitInstanceBindRegionCount is not 0, pSplitInstanceBindRegions must be a valid pointer to an array of splitInstanceBindRegionCount VkRect2D structures

If the pNext chain of VkBindImageMemoryInfo includes a VkBindImageMemorySwapchainInfoKHR structure, then that structure includes a swapchain handle and image index indicating that the image will be bound to memory from that swapchain.

The VkBindImageMemorySwapchainInfoKHR structure is defined as:

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with
// VK_KHR_swapchain
typedef struct VkBindImageMemorySwapchainInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
    uint32_t           imageIndex;
} VkBindImageMemorySwapchainInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- swapchain is VK_NULL_HANDLE or a swapchain handle.
- imageIndex is an image index within swapchain.

If swapchain is not NULL, the swapchain and imageIndex are used to determine the memory that the image is bound to, instead of memory and memoryOffset.

Memory can be bound to a swapchain and use the pDeviceIndices or pSplitInstanceBindRegions members of VkBindImageMemoryDeviceGroupInfo.

Valid Usage

- VUID-VkBindImageMemorySwapchainInfoKHR-imageIndex-01644
  
  imageIndex must be less than the number of images in swapchain
Valid Usage (Implicit)

- VUID-VkBindImageMemorySwapchainInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR

- VUID-VkBindImageMemorySwapchainInfoKHR-swapchain-parameter
  swapchain must be a valid VkSwapchainKHR handle

Host Synchronization

- Host access to swapchain must be externally synchronized

In order to bind planes of a disjoint image, add a VkBindImagePlaneMemoryInfo structure to the pNext chain of VkBindImageMemoryInfo.

The VkBindImagePlaneMemoryInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkBindImagePlaneMemoryInfo {
    VkStructureType         sType;
    const void*              pNext;
    VkImageAspectFlagBits    planeAspect;
} VkBindImagePlaneMemoryInfo;
```

or the equivalent

```c
// Provided by VK_KHR_sampler_yccbcr_conversion
typedef VkBindImagePlaneMemoryInfo VkBindImagePlaneMemoryInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- planeAspect is the aspect of the disjoint image plane to bind.
Valid Usage

- **VUID-VkBindImagePlaneMemoryInfo-planeAspect-02283**
  If the image’s tiling is VK_IMAGE_TILING_LINEAR or VK_IMAGE_TILING_OPTIMAL, then planeAspect must be a single valid format plane for the image (that is, for a two-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT, and for a three-plane image planeAspect must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT or VK_IMAGE_ASPECT_PLANE_2_BIT).

- **VUID-VkBindImagePlaneMemoryInfo-planeAspect-02284**
  If the image’s tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then planeAspect must be a single valid memory plane for the image (that is, aspectMask must specify a plane index that is less than the VkDrmFormatModifierPropertiesEXT::drmFormatModifierPlaneCount associated with the image’s format and VkImageDrmFormatModifierPropertiesEXT::drmFormatModifier).

Valid Usage (Implicit)

- **VUID-VkBindImagePlaneMemoryInfo-sType-sType**
  sType must be VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO

- **VUID-VkBindImagePlaneMemoryInfo-planeAspect-parameter**
  planeAspect must be a valid VkImageAspectFlagBits value

Buffer-Image Granularity

There is an implementation-dependent limit, bufferImageGranularity, which specifies a page-like granularity at which linear and non-linear resources must be placed in adjacent memory locations to avoid aliasing. Two resources which do not satisfy this granularity requirement are said to alias. bufferImageGranularity is specified in bytes, and must be a power of two. Implementations which do not impose a granularity restriction may report a bufferImageGranularity value of one.

**Note**

Despite its name, bufferImageGranularity is really a granularity between “linear” and “non-linear” resources.

Given resourceA at the lower memory offset and resourceB at the higher memory offset in the same VkDeviceMemory object, where one resource is linear and the other is non-linear (as defined in the Glossary), and the following:

- resourceA.end = resourceA.memoryOffset + resourceA.size - 1
- resourceA.endPage = resourceA.end & ~(bufferImageGranularity-1)
- resourceB.start = resourceB.memoryOffset
- resourceB.startPage = resourceB.start & ~(bufferImageGranularity-1)

The following property must hold:
That is, the end of the first resource (A) and the beginning of the second resource (B) must be on separate “pages” of size \texttt{bufferImageGranularity}. \texttt{bufferImageGranularity} may be different than the physical page size of the memory heap. This restriction is only needed when a linear resource and a non-linear resource are adjacent in memory and will be used simultaneously. The memory ranges of adjacent resources can be closer than \texttt{bufferImageGranularity}, provided they meet the alignment requirement for the objects in question.

Sparse block size in bytes and sparse image and buffer memory alignments must all be multiples of the \texttt{bufferImageGranularity}. Therefore, memory bound to sparse resources naturally satisfies the \texttt{bufferImageGranularity}.

### 12.8. Resource Sharing Mode

Buffer and image objects are created with a \textit{sharing mode} controlling how they can be accessed from queues. The supported sharing modes are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

- **\texttt{VK_SHARING_MODE_EXCLUSIVE}** specifies that access to any range or image subresource of the object will be exclusive to a single queue family at a time.
- **\texttt{VK_SHARING_MODE_CONCURRENT}** specifies that concurrent access to any range or image subresource of the object from multiple queue families is supported.

\textbf{Note} \texttt{VK_SHARING_MODE_CONCURRENT} may result in lower performance access to the buffer or image than \texttt{VK_SHARING_MODE_EXCLUSIVE}.

Ranges of buffers and image subresources of image objects created using **\texttt{VK_SHARING_MODE_EXCLUSIVE}** must only be accessed by queues in the queue family that has \textit{ownership} of the resource. Upon creation, such resources are not owned by any queue family; ownership is implicitly acquired upon first use within a queue. Once a resource using **\texttt{VK_SHARING_MODE_EXCLUSIVE}** is owned by some queue family, the application must perform a queue family ownership transfer to make the memory contents of a range or image subresource accessible to a different queue family.

\textbf{Note} Images still require a layout transition from **\texttt{VK_IMAGE_LAYOUT_UNDEFINED}** or **\texttt{VK_IMAGE_LAYOUT_PREINITIALIZED}** before being used on the first queue.

A queue family can take ownership of an image subresource or buffer range of a resource created
with `VK_SHARING_MODE_EXCLUSIVE`, without an ownership transfer, in the same way as for a resource that was just created; however, taking ownership in this way has the effect that the contents of the image subresource or buffer range are undefined.

Ranges of buffers and image subresources of image objects created using `VK_SHARING_MODE_CONCURRENT` must only be accessed by queues from the queue families specified through the `queueFamilyIndexCount` and `pQueueFamilyIndices` members of the corresponding create info structures.

### 12.8.1. External Resource Sharing

Resources should only be accessed in the Vulkan instance that has exclusive ownership of their underlying memory. Only one Vulkan instance has exclusive ownership of a resource's underlying memory at a given time, regardless of whether the resource was created using `VK_SHARING_MODE_EXCLUSIVE` or `VK_SHARING_MODE_CONCURRENT`. Applications can transfer ownership of a resource's underlying memory only if the memory has been imported from or exported to another instance or external API using external memory handles. The semantics for transferring ownership outside of the instance are similar to those used for transferring ownership of `VK_SHARING_MODE_EXCLUSIVE` resources between queues, and is also accomplished using `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` operations. Applications must

1. Release exclusive ownership from the source instance or API.
2. Ensure the release operation has completed using semaphores or fences.
3. Acquire exclusive ownership in the destination instance or API

Unlike queue ownership transfers, the destination instance or API is not specified explicitly when releasing ownership, nor is the source instance or API specified when acquiring ownership. Instead, the image or memory barrier's `dstQueueFamilyIndex` or `srcQueueFamilyIndex` parameters are set to the reserved queue family index `VK_QUEUE_FAMILY_EXTERNAL` or `VK_QUEUE_FAMILY_FOREIGN_EXT` to represent the external destination or source respectively.

Binding a resource to a memory object shared between multiple Vulkan instances or other APIs does not change the ownership of the underlying memory. The first entity to access the resource implicitly acquires ownership. Accessing a resource backed by memory that is owned by a particular instance or API has the same semantics as accessing a `VK_SHARING_MODE_EXCLUSIVE` resource, with one exception: Implementations must ensure layout transitions performed on one member of a set of identical subresources of identical images that alias the same range of an underlying memory object affect the layout of all the subresources in the set.

As a corollary, writes to any image subresources in such a set must not make the contents of memory used by other subresources in the set undefined. An application can define the content of a subresource of one image by performing device writes to an identical subresource of another image provided both images are bound to the same region of external memory. Applications may also add resources to such a set after the content of the existing set members has been defined without making the content undefined by creating a new image with the initial layout `VK_IMAGE_LAYOUT_UNDEFINED` and binding it to the same region of external memory as the existing images.
Because layout transitions apply to all identical images aliasing the same region of external memory, the actual layout of the memory backing a new image as well as an existing image with defined content will not be undefined. Such an image is not usable until it acquires ownership of its memory from the existing owner. Therefore, the layout specified as part of this transition will be the true initial layout of the image. The undefined layout specified when creating it is a placeholder to simplify valid usage requirements.

12.9. Memory Aliasing

A range of a VkDeviceMemory allocation is aliased if it is bound to multiple resources simultaneously, as described below, via vkBindImageMemory, vkBindBufferMemory, vkBindAccelerationStructureMemoryNV, via sparse memory bindings, or by binding the memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms.

Consider two resources, resource\textsubscript{A} and resource\textsubscript{B}, bound respectively to memory range\textsubscript{A} and range\textsubscript{B}. Let paddedRange\textsubscript{A} and paddedRange\textsubscript{B} be, respectively, range\textsubscript{A} and range\textsubscript{B} aligned to bufferImageGranularity. If the resources are both linear or both non-linear (as defined in the Glossary), then the resources alias the memory in the intersection of range\textsubscript{A} and range\textsubscript{B}. If one resource is linear and the other is non-linear, then the resources alias the memory in the intersection of paddedRange\textsubscript{A} and paddedRange\textsubscript{B}.

Applications can alias memory, but use of multiple aliases is subject to several constraints.

Memory aliasing can be useful to reduce the total device memory footprint of an application, if some large resources are used for disjoint periods of time.

When a non-linear, non-VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT image is bound to an aliased range, all image subresources of the image overlap the range. When a linear image is bound to an aliased range, the image subresources that (according to the image’s advertised layout) include bytes from the aliased range overlap the range. When a VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT image has sparse image blocks bound to an aliased range, only image subresources including those sparse image blocks overlap the range, and when the memory bound to the image’s mip tail overlaps an aliased range all image subresources in the mip tail overlap the range.

Buffers, and linear image subresources in either the VK_IMAGE_LAYOUT_PREINITIALIZED or VK_IMAGE_LAYOUT_GENERAL layouts, are host-accessible subresources. That is, the host has a well-defined addressing scheme to interpret the contents, and thus the layout of the data in memory can be consistently interpreted across aliases if each of those aliases is a host-accessible subresource. Non-linear images, and linear image subresources in other layouts, are not host-accessible.

If two aliases are both host-accessible, then they interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

If two aliases are both images that were created with identical creation parameters, both were
created with the `VK_IMAGE_CREATE_ALIAS_BIT` flag set, and both are bound identically to memory except for `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` and `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions`, then they interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

Additionally, if an individual plane of a multi-planar image and a single-plane image alias the same memory, then they also interpret the contents of the memory in consistent ways under the same conditions, but with the following modifications:

- Both must have been created with the `VK_IMAGE_CREATE_DISJOINT_BIT` flag.
- The single-plane image must have a `VkFormat` that is equivalent to that of the multi-planar image’s individual plane.
- The single-plane image and the individual plane of the multi-planar image must be bound identically to memory except for `VkBindImageMemoryDeviceGroupInfo::pDeviceIndices` and `VkBindImageMemoryDeviceGroupInfo::pSplitInstanceBindRegions`.
- The width and height of the single-plane image are derived from the multi-planar image’s dimensions in the manner listed for plane compatibility for the aliased plane.
- If either image’s tiling is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`, then both images must be linear.
- All other creation parameters must be identical.

Aliases created by binding the same memory to resources in multiple Vulkan instances or external APIs using external memory handle export and import mechanisms interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

Otherwise, the aliases interpret the contents of the memory differently, and writes via one alias make the contents of memory partially or completely undefined to the other alias. If the first alias is a host-accessible subresource, then the bytes affected are those written by the memory operations according to its addressing scheme. If the first alias is not host-accessible, then the bytes affected are those overlapped by the image subresources that were written. If the second alias is a host-accessible subresource, the affected bytes become undefined. If the second alias is a non-host-accessible, all sparse image blocks (for sparse partially-resident images) or all image subresources (for non-sparse image and fully resident sparse images) that overlap the affected bytes become undefined.

If any image subresources are made undefined due to writes to an alias, then each of those image subresources must have its layout transitioned from `VK_IMAGE_LAYOUT_UNDEFINED` to a valid layout before it is used, or from `VK_IMAGE_LAYOUT_PREINITIALIZED` if the memory has been written by the host. If any sparse blocks of a sparse image have been made undefined, then only the image subresources containing them must be transitioned.

Use of an overlapping range by two aliases must be separated by a memory dependency using the appropriate access types if at least one of those uses performs writes, whether the aliases interpret memory consistently or not. If buffer or image memory barriers are used, the scope of the barrier must contain the entire range and/or set of image subresources that overlap.

If two aliasing image views are used in the same framebuffer, then the render pass must declare
the attachments using the `VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT`, and follow the other rules listed in that section.

**Note**
Memory recycled via an application suballocator (i.e. without freeing and reallocating the memory objects) is not substantially different from memory aliasing. However, a suballocator usually waits on a fence before recycling a region of memory, and signaling a fence involves sufficient implicit dependencies to satisfy all the above requirements.
Chapter 13. Samplers

**VkSampler** objects represent the state of an image sampler which is used by the implementation to read image data and apply filtering and other transformations for the shader.

Samplers are represented by **VkSampler** handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSampler)
```

To create a sampler object, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateSampler(  
    VkDevice                                    device,  
    const VkSamplerCreateInfo*                  pCreateInfo,  
    const VkAllocationCallbacks*                pAllocator,  
    VkSampler*                                  pSampler);
```

- **device** is the logical device that creates the sampler.
- **pCreateInfo** is a pointer to a **VkSamplerCreateInfo** structure specifying the state of the sampler object.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pSampler** is a pointer to a **VkSampler** handle in which the resulting sampler object is returned.

 Valid Usage

- **VUID-vkCreateSampler-maxSamplerAllocationCount-04110**

  There must be less than **VkPhysicalDeviceLimits::maxSamplerAllocationCount** **VkSampler** objects currently created on the device.
Valid Usage (Implicit)

- **VUID-vkCreateSampler-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkCreateSampler-pCreateInfo-parameter**
  
  `pCreateInfo` must be a valid pointer to a valid `VkSamplerCreateInfo` structure

- **VUID-vkCreateSampler-pAllocator-parameter**
  
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateSampler-pSampler-parameter**
  
  `pSampler` must be a valid pointer to a `VkSampler` handle

Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkSamplerCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSamplerCreateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkSamplerCreateFlags    flags;
    VkFilter                magFilter;
    VkFilter                minFilter;
    VkSamplerMipmapMode     mipmapMode;
    VkSamplerAddressMode    addressModeU;
    VkSamplerAddressMode    addressModeV;
    VkSamplerAddressMode    addressModeW;
    float                    mipLodBias;
    VkBool32                 anisotropyEnable;
    float                    maxAnisotropy;
    VkBool32                 compareEnable;
    VkCompareOp              compareOp;
    float                    minLod;
    float                    maxLod;
    VkBorderColor           borderColor;
    VkBool32                 unnormalizedCoordinates;
} VkSamplerCreateInfo;
```
• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **flags** is a bitmask of **VkSamplerCreateFlagBits** describing additional parameters of the sampler.

• **magFilter** is a **VkFilter** value specifying the magnification filter to apply to lookups.

• **minFilter** is a **VkFilter** value specifying the minification filter to apply to lookups.

• **mipmapMode** is a **VkSamplerMipmapMode** value specifying the mipmap filter to apply to lookups.

• **addressModeU** is a **VkSamplerAddressMode** value specifying the addressing mode for outside [0..1] range for U coordinate.

• **addressModeV** is a **VkSamplerAddressMode** value specifying the addressing mode for outside [0..1] range for V coordinate.

• **addressModeW** is a **VkSamplerAddressMode** value specifying the addressing mode for outside [0..1] range for W coordinate.

• **mipLodBias** is the bias to be added to mipmap LOD (level-of-detail) calculation and bias provided by image sampling functions in SPIR-V, as described in the **Level-of-Detail Operation** section.

• **anisotropyEnable** is **VK_TRUE** to enable anisotropic filtering, as described in the **Texel Anisotropic Filtering** section, or **VK_FALSE** otherwise.

• **maxAnisotropy** is the anisotropy value clamp used by the sampler when **anisotropyEnable** is **VK_TRUE**. If **anisotropyEnable** is **VK_FALSE**, **maxAnisotropy** is ignored.

• **compareEnable** is **VK_TRUE** to enable comparison against a reference value during lookups, or **VK_FALSE** otherwise.

  ◦ Note: Some implementations will default to shader state if this member does not match.

• **compareOp** is a **VkCompareOp** value specifying the comparison function to apply to fetched data before filtering as described in the **Depth Compare Operation** section.

• **minLod** is used to clamp the minimum of the computed LOD value.

• **maxLod** is used to clamp the maximum of the computed LOD value. To avoid clamping the maximum value, set **maxLod** to the constant **VK_LOD_CLAMP_NONE**.

• **borderColor** is a **VkBorderColor** value specifying the predefined border color to use.

• **unnormalizedCoordinates** controls whether to use unnormalized or normalized texel coordinates to address texels of the image. When set to **VK_TRUE**, the range of the image coordinates used to lookup the texel is in the range of zero to the image dimensions for x, y and z. When set to **VK_FALSE** the range of image coordinates is zero to one.

When **unnormalizedCoordinates** is **VK_TRUE**, images the sampler is used with in the shader have the following requirements:

  ◦ The **viewType** must be either **VK_IMAGE_VIEW_TYPE_1D** or **VK_IMAGE_VIEW_TYPE_2D**.

  ◦ The image view must have a single layer and a single mip level.

When **unnormalizedCoordinates** is **VK_TRUE**, image built-in functions in the shader that use the sampler have the following requirements:

  ◦ The functions must not use projection.
The functions **must** not use offsets.

### Mapping of OpenGL to Vulkan filter modes

*magFilter* values of `VK_FILTER_NEAREST` and `VK_FILTER_LINEAR` directly correspond to `GL_NEAREST` and `GL_LINEAR` magnification filters. *minFilter* and *mipmapMode* combine to correspond to the similarly named OpenGL minification filter of `GL_minFilter_MIPMAP_mipmapMode` (e.g. *minFilter* of `VK_FILTER_LINEAR` and *mipmapMode* of `VK_SAMPLER_MIPMAP_MODE_NEAREST` correspond to `GL_LINEAR_MIPMAP_NEAREST`).

There are no Vulkan filter modes that directly correspond to OpenGL minification filters of `GL_LINEAR` or `GL_NEAREST`, but they can be emulated using `VK_SAMPLER_MIPMAP_MODE_NEAREST, minLod = 0, and maxLod = 0.25, and using minFilter = VK_FILTER_LINEAR or minFilter = VK_FILTER_NEAREST, respectively.`

Note that using a *maxLod* of zero would cause magnification to always be performed, and the *magFilter* to always be used. This is valid, just not an exact match for OpenGL behavior. Clamping the maximum LOD to 0.25 allows the λ value to be non-zero and minification to be performed, while still always rounding down to the base level. If the *minFilter* and *magFilter* are equal, then using a *maxLod* of zero also works.

The maximum number of sampler objects which can be simultaneously created on a device is implementation-dependent and specified by the `maxSamplerAllocationCount` member of the `VkPhysicalDeviceLimits` structure.

**Note**

For historical reasons, if `maxSamplerAllocationCount` is exceeded, some implementations may return `VK_ERROR_TOO_MANY_OBJECTS`. Exceeding this limit will result in undefined behavior, and an application should not rely on the use of the returned error code in order to identify when the limit is reached.

Since `VkSampler` is a non-dispatchable handle type, implementations may return the same handle for sampler state vectors that are identical. In such cases, all such objects would only count once against the `maxSamplerAllocationCount` limit.
Valid Usage

- **VUID-VkSamplerCreateInfo-mipLodBias-01069**
  
  The absolute value of `mipLodBias` must be less than or equal to `VkPhysicalDeviceLimits::maxSamplerLodBias`

- **VUID-VkSamplerCreateInfo-samplerMipLodBias-04467**
  
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::samplerMipLodBias` is `VK_FALSE`, `mipLodBias` must be zero.

- **VUID-VkSamplerCreateInfo-maxLod-01973**
  
  `maxLod` must be greater than or equal to `minLod`

- **VUID-VkSamplerCreateInfo-anisotropyEnable-01070**
  
  If the anisotropic sampling feature is not enabled, `anisotropyEnable` must be `VK_FALSE`

- **VUID-VkSamplerCreateInfo-anisotropyEnable-01071**
  
  If `anisotropyEnable` is `VK_TRUE`, `maxAnisotropy` must be between 1.0 and `VkPhysicalDeviceLimits::maxSamplerAnisotropy`, inclusive

- **VUID-VkSamplerCreateInfo-minFilter-01645**
  
  If sampler `Y'CBCR` conversion is enabled and the potential format features of the sampler `Y'CBCR` conversion do not support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT`, `minFilter` and `magFilter` must be equal to the sampler `Y'CBCR` conversion's chromaFilter

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01072**
  
  If `unnormalizedCoordinates` is `VK_TRUE`, `minFilter` and `magFilter` must be equal

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01073**
  
  If `unnormalizedCoordinates` is `VK_TRUE`, `mipmapMode` must be `VK_SAMPLER_MIPMAP_MODE_NEAREST`

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01074**
  
  If `unnormalizedCoordinates` is `VK_TRUE`, `minLod` and `maxLod` must be zero

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01075**
  
  If `unnormalizedCoordinates` is `VK_TRUE`, `addressModeU` and `addressModeV` must each be either `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` or `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01076**
  
  If `unnormalizedCoordinates` is `VK_TRUE`, `anisotropyEnable` must be `VK_FALSE`

- **VUID-VkSamplerCreateInfo-unnormalizedCoordinates-01077**
  
  If `unnormalizedCoordinates` is `VK_TRUE`, `compareEnable` must be `VK_FALSE`

- **VUID-VkSamplerCreateInfo-addressModeU-01078**
  
  If any of `addressModeU`, `addressModeV` or `addressModeW` are `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`, `borderColor` must be a valid `VkBorderColor` value

- **VUID-VkSamplerCreateInfo-addressModeU-01646**
  
  If sampler `Y'CBCR` conversion is enabled, `addressModeU`, `addressModeV`, and `addressModeW` must be `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`, `anisotropyEnable` must be `VK_FALSE`, and `unnormalizedCoordinates` must be `VK_FALSE`
The sampler reduction mode must be set to VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE if sampler Y’CbCr conversion is enabled.

If samplerMirrorClampToEdge is not enabled, and if the VK_KHR_sampler_mirror_clamp_to_edge extension is not enabled, addressModeU, addressModeV and addressModeW must not be VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE.

If compareEnable is VK_TRUE, compareOp must be a valid VkCompareOp value.

If either magFilter or minFilter is VK_FILTER_CUBIC_EXT, anisotropyEnable must be VK_FALSE.

If compareEnable is VK_TRUE, the reductionMode member of VkSamplerReductionModeCreateInfo must be VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then minFilter and magFilter must be equal.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then mipmapMode must be VK_SAMPLER_MIPMAP_MODE_NEAREST.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then minLod and maxLod must be zero.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then addressModeU and addressModeV must each be either VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE or VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then anisotropyEnable must be VK_FALSE.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then compareEnable must be VK_FALSE.

If flags includes VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT, then unnormalizedCoordinates must be VK_FALSE.

If borderColor is one of VK_BORDER_COLOR_FLOAT_CUSTOM_EXT or VK_BORDER_COLOR_INT_CUSTOM_EXT, then a VkSamplerCustomBorderColorCreateInfoEXT must be present in the pNext chain.

If the customBorderColors feature is not enabled, borderColor must not be VK_BORDER_COLOR_FLOAT_CUSTOM_EXT or VK_BORDER_COLOR_INT_CUSTOM_EXT.
If `borderColor` is one of `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT`, and `VkSamplerCustomBorderColorCreateInfoEXT::format` is not `VK_FORMAT_UNDEFINED`, `VkSamplerCustomBorderColorCreateInfoEXT::customBorderColor` must be within the range of values representable in `format`.

The maximum number of samplers with custom border colors which can be simultaneously created on a device is implementation-dependent and specified by the `maxCustomBorderColorSamplers` member of the `VkPhysicalDeviceCustomBorderColorPropertiesEXT` structure.

### Valid Usage (Implicit)

- **sType must be** `VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO`
- **pNext chain** must be either `NULL` or a pointer to a valid instance of `VkSamplerCustomBorderColorCreateInfoEXT`, `VkSamplerReductionModeCreateInfo`, or `VkSamplerYcbcrConversionInfo`
- **The sType value of each struct in the pNext chain must be unique**
- **flags** must be a valid combination of `VkSamplerCreateFlagBits` values
- **magFilter** must be a valid `VkFilter` value
- **minFilter** must be a valid `VkFilter` value
- **mipmapMode** must be a valid `VkSamplerMipmapMode` value
- **addressModeU** must be a valid `VkSamplerAddressMode` value
- **addressModeV** must be a valid `VkSamplerAddressMode` value
- **addressModeW** must be a valid `VkSamplerAddressMode` value

Bits which can be set in `VkSamplerCreateInfo::flags`, specifying additional parameters of a sampler, are:
typedef enum VkSamplerCreateFlagBits {
    VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT = 0x00000001,
    VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT = 0x00000002,
} VkSamplerCreateFlagBits;

• **VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT** specifies that the sampler will read from an image created with flags containing **VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT**.

• **VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT** specifies that the implementation may use approximations when reconstructing a full color value for texture access from a subsampled image.

Note

The approximations used when **VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT** is specified are implementation defined. Some implementations may interpolate between fragment density levels in a subsampled image. In that case, this bit may be used to decide whether the interpolation factors are calculated per fragment or at a coarser granularity.

typedef VkFlags VkSamplerCreateFlags;

**VkSamplerCreateFlags** is a bitmask type for setting a mask of zero or more **VkSamplerCreateFlagBits**.

The **VkSamplerReductionModeCreateInfo** structure is defined as:

typedef struct VkSamplerReductionModeCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkSamplerReductionMode reductionMode;
} VkSamplerReductionModeCreateInfo;

or the equivalent

// Provided by VK_EXT_sampler_filter_minmax
typedef VkSamplerReductionModeCreateInfo VkSamplerReductionModeCreateInfoEXT;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **reductionMode** is a **VkSamplerReductionMode** value controlling how texture filtering combines
If the \texttt{pNext} chain of \texttt{VkSamplerCreateInfo} includes a \texttt{VkSamplerReductionModeCreateInfo} structure, then that structure includes a mode that controls how texture filtering combines texel values.

If this structure is not present, \texttt{reductionMode} is considered to be \texttt{VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE}.

### Valid Usage (Implicit)

- \texttt{VUID-VkSamplerReductionModeCreateInfo-sType-sType}  
  \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO}

- \texttt{VUID-VkSamplerReductionModeCreateInfo-reductionMode-parameter}  
  \texttt{reductionMode} must be a valid \texttt{VkSamplerReductionMode} value

Reduction modes are specified by \texttt{VkSamplerReductionMode}, which takes values:

```c
typedef enum VkSamplerReductionMode {
    VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE = 0,
    VK_SAMPLER_REDUCTION_MODE_MIN = 1,
    VK_SAMPLER_REDUCTION_MODE_MAX = 2,
    // Provided by VK_EXT_sampler_filter_minmax
    VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT = VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE,
    // Provided by VK_EXT_sampler_filter_minmax
    VK_SAMPLER_REDUCTION_MODE_MIN_EXT = VK_SAMPLER_REDUCTION_MODE_MIN,
    // Provided by VK_EXT_sampler_filter_minmax
    VK_SAMPLER_REDUCTION_MODE_MAX_EXT = VK_SAMPLER_REDUCTION_MODE_MAX,
} VkSamplerReductionMode;
```

or the equivalent

```c
// Provided by VK_EXT_sampler_filter_minmax
typedef VkSamplerReductionMode VkSamplerReductionModeEXT;
```

- \texttt{VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE} specifies that texel values are combined by computing a weighted average of values in the footprint, using weights as specified in the image operations chapter.

- \texttt{VK_SAMPLER_REDUCTION_MODE_MIN} specifies that texel values are combined by taking the component-wise minimum of values in the footprint with non-zero weights.

- \texttt{VK_SAMPLER_REDUCTION_MODE_MAX} specifies that texel values are combined by taking the component-wise maximum of values in the footprint with non-zero weights.

Possible values of the \texttt{VkSamplerCreateInfo::magFilter} and \texttt{minFilter} parameters, specifying filters used for texture lookups, are:
typedef enum VkFilter {
    VK_FILTER_NEAREST = 0,
    VK_FILTER_LINEAR = 1,
    // Provided by VK_IMG_filter_cubic
    VK_FILTER_CUBIC_IMG = 100015000,
    // Provided by VK_EXT_filter_cubic
    VK_FILTER_CUBIC_EXT = VK_FILTER_CUBIC_IMG,
} VkFilter;

- **VK_FILTER_NEAREST** specifies nearest filtering.
- **VK_FILTER_LINEAR** specifies linear filtering.
- **VK_FILTER_CUBIC_EXT** specifies cubic filtering.

These filters are described in detail in *Texel Filtering*.

Possible values of the `VkSamplerCreateInfo::mipmapMode`, specifying the mipmap mode used for texture lookups, are:

typedef enum VkSamplerMipmapMode {
    VK_SAMPLER_MIPMAP_MODE_NEAREST = 0,
    VK_SAMPLER_MIPMAP_MODE_LINEAR = 1,
} VkSamplerMipmapMode;

- **VK_SAMPLER_MIPMAP_MODE_NEAREST** specifies nearest filtering.
- **VK_SAMPLER_MIPMAP_MODE_LINEAR** specifies linear filtering.

These modes are described in detail in *Texel Filtering*.

Possible values of the `VkSamplerCreateInfo::addressMode*` parameters, specifying the behavior of sampling with coordinates outside the range [0,1] for the respective u, v, or w coordinate as defined in the *Wrapping Operation* section, are:

typedef enum VkSamplerAddressMode {
    VK_SAMPLER_ADDRESS_MODE_REPEAT = 0,
    VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT = 1,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE = 2,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER = 3,
    // Provided by VK_KHR_sampler_mirror_clamp_to_edge
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE = 4,
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE_KHR =
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE,
} VkSamplerAddressMode;
- `VK_SAMPLER_ADDRESS_MODE_REPEAT` specifies that the repeat wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT` specifies that the mirrored repeat wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` specifies that the clamp to edge wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER` specifies that the clamp to border wrap mode will be used.
- `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` specifies that the mirror clamp to edge wrap mode will be used. This is only valid if the `VK_KHR_sampler_mirror_clamp_to_edge` extension is enabled.

Possible values of `VkSamplerCreateInfo::borderColor`, specifying the border color used for texture lookups, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkBorderColor {
    VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK = 0,
    VK_BORDER_COLOR_INT_TRANSPARENT_BLACK = 1,
    VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK = 2,
    VK_BORDER_COLOR_INT_OPAQUE_BLACK = 3,
    VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE = 4,
    VK_BORDER_COLOR_INT_OPAQUE_WHITE = 5,
    // Provided by VK_EXT_custom_border_color
    VK_BORDER_COLOR_FLOAT_CUSTOM_EXT = 1000287003,
    // Provided by VK_EXT_custom_border_color
    VK_BORDER_COLOR_INT_CUSTOM_EXT = 1000287004,
} VkBorderColor;
```

- `VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK` specifies a transparent, floating-point format, black color.
- `VK_BORDER_COLOR_INT_TRANSPARENT_BLACK` specifies a transparent, integer format, black color.
- `VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK` specifies an opaque, floating-point format, black color.
- `VK_BORDER_COLOR_INT_OPAQUE_BLACK` specifies an opaque, integer format, black color.
- `VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE` specifies an opaque, floating-point format, white color.
- `VK_BORDER_COLOR_INT_OPAQUE_WHITE` specifies an opaque, integer format, white color.
- `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` indicates that a `VkSamplerCustomBorderColorCreateInfoEXT` structure is present in the `VkSamplerCreateInfo::pNext` chain which contains the color data in floating-point format.
- `VK_BORDER_COLOR_INT_CUSTOM_EXT` indicates that a `VkSamplerCustomBorderColorCreateInfoEXT` structure is present in the `VkSamplerCreateInfo::pNext` chain which contains the color data in integer format.

These colors are described in detail in `Texel Replacement`.

To destroy a sampler, call:
// Provided by VK_VERSION_1_0
void vkDestroySampler(
    VkDevice                                    device,
    VkSampler                                   sampler,
    const VkAllocationCallbacks*                pAllocator);

- **device** is the logical device that destroys the sampler.
- **sampler** is the sampler to destroy.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroySampler-sampler-01082**
  All submitted commands that refer to **sampler** must have completed execution

- **VUID-vkDestroySampler-sampler-01083**
  If **VkAllocationCallbacks** were provided when **sampler** was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroySampler-sampler-01084**
  If no **VkAllocationCallbacks** were provided when **sampler** was created, **pAllocator** must be NULL

### Valid Usage (Implicit)

- **VUID-vkDestroySampler-device-parameter**
  **device** must be a valid **VkDevice** handle

- **VUID-vkDestroySampler-sampler-parameter**
  If **sampler** is not **VK_NULL_HANDLE**, **sampler** must be a valid **VkSampler** handle

- **VUID-vkDestroySampler-pAllocator-parameter**
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- **VUID-vkDestroySampler-sampler-parent**
  If **sampler** is a valid handle, it must have been created, allocated, or retrieved from **device**

### Host Synchronization

- Host access to **sampler** must be externally synchronized
13.1. Sampler Y'CbCr conversion

To create a sampler with Y'CbCr conversion enabled, add a VkSamplerYcbcrConversionInfo structure to the pNext chain of the VkSamplerCreateInfo structure. To create a sampler Y'CbCr conversion, the samplerYcbcrConversion feature must be enabled. Conversion must be fixed at pipeline creation time, through use of a combined image sampler with an immutable sampler in VkDescriptorSetLayoutBinding.

A VkSamplerYcbcrConversionInfo must be provided for samplers to be used with image views that access VK_IMAGE_ASPECT_COLOR_BIT if the format appears in Formats requiring sampler Y'CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views, or if the image view has an external format.

The VkSamplerYcbcrConversionInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionInfo {
    VkStructureType sType;
    const void* pNext;
    VkSamplerYcbcrConversion conversion;
} VkSamplerYcbcrConversionInfo;
```

or the equivalent

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkSamplerYcbcrConversionInfo VkSamplerYcbcrConversionInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **conversion** is a VkSamplerYcbcrConversion handle created with vkCreateSamplerYcbcrConversion.

Valid Usage (Implicit)

- VUID-VkSamplerYcbcrConversionInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO
- VUID-VkSamplerYcbcrConversionInfo-conversion-parameter
  conversion must be a valid VkSamplerYcbcrConversion handle

A sampler Y'CbCr conversion is an opaque representation of a device-specific sampler Y'CbCr conversion description, represented as a VkSamplerYcbcrConversion handle:
or the equivalent

```cpp
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkSamplerYcbcrConversion VkSamplerYcbcrConversionKHR;
```

To create a `VkSamplerYcbcrConversion`, call:

```cpp
// Provided by VK_VERSION_1_1
VkResult vkCreateSamplerYcbcrConversion(
    VkDevice                                    device,
    const VkSamplerYcbcrConversionCreateInfo*   pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSamplerYcbcrConversion*                   pYcbcrConversion);
```

or the equivalent command

```cpp
// Provided by VK_KHR_sampler_ycbcr_conversion
VkResult vkCreateSamplerYcbcrConversionKHR(
    VkDevice                                    device,
    const VkSamplerYcbcrConversionCreateInfo*   pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSamplerYcbcrConversion*                   pYcbcrConversion);
```

- `device` is the logical device that creates the sampler Y’C_{A}C_{B}C_{R} conversion.
- `pCreateInfo` is a pointer to a `VkSamplerYcbcrConversionCreateInfo` structure specifying the requested sampler Y’C_{A}C_{B}C_{R} conversion.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pYcbcrConversion` is a pointer to a `VkSamplerYcbcrConversion` handle in which the resulting sampler Y’C_{A}C_{B}C_{R} conversion is returned.

The interpretation of the configured sampler Y’C_{A}C_{B}C_{R} conversion is described in more detail in the description of sampler Y’C_{A}C_{B}C_{R} conversion in the Image Operations chapter.

### Valid Usage

- **VUID-vkCreateSamplerYcbcrConversion-None-01648**

  The sampler Y’C_{A}C_{B}C_{R} conversion feature must be enabled
Valid Usage (Implicit)

- VUID-vkCreateSamplerYcbcrConversion-device-parameter
  device must be a valid VkDevice handle

- VUID-vkCreateSamplerYcbcrConversion-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkSamplerYcbcrConversionCreateInfo structure

- VUID-vkCreateSamplerYcbcrConversion-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateSamplerYcbcrConversion-pYcbcrConversion-parameter
  pYcbcrConversion must be a valid pointer to a VkSamplerYcbcrConversion handle

Return Codes

Success

  - VK_SUCCESS

Failure

  - VK_ERROR_OUT_OF_HOST_MEMORY
  - VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkSamplerYcbcrConversionCreateInfo structure is defined as:

```
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkSamplerYcbcrModelConversion ycbcrModel;
    VkSamplerYcbcrRange ycbcrRange;
    VkComponentMapping components;
    VkChromaLocation xChromaOffset;
    VkChromaLocation yChromaOffset;
    VkFilter chromaFilter;
    VkBool32 forceExplicitReconstruction;
} VkSamplerYcbcrConversionCreateInfo;
```

or the equivalent

```
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkSamplerYcbcrConversionCreateInfo VkSamplerYcbcrConversionCreateInfoKHR;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **format** is the format of the image from which color information will be retrieved.

• **ycbcrModel** describes the color matrix for conversion between color models.

• **ycbcrRange** describes whether the encoded values have headroom and foot room, or whether the encoding uses the full numerical range.

• **components** applies a *swizzle* based on **VkComponentSwizzle** enums prior to range expansion and color model conversion.

• **xChromaOffset** describes the **sample location** associated with downsampled chroma channels in the x dimension. **xChromaOffset** has no effect for formats in which chroma channels are not downsampled horizontally.

• **yChromaOffset** describes the **sample location** associated with downsampled chroma channels in the y dimension. **yChromaOffset** has no effect for formats in which the chroma channels are not downsampled vertically.

• **chromaFilter** is the filter for chroma reconstruction.

• **forceExplicitReconstruction** can be used to ensure that reconstruction is done explicitly, if supported.

  
  ![](image)

  **Note**
  Setting **forceExplicitReconstruction** to **VK_TRUE** may have a performance penalty on implementations where explicit reconstruction is not the default mode of operation.

  If **format** supports **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT** the **forceExplicitReconstruction** value behaves as if it was set to **VK_TRUE**.

If the **pNext** chain includes a **VkExternalFormatANDROID** structure with non-zero **externalFormat** member, the sampler **Y’C_bC_r** conversion object represents an **external format conversion**, and **format** must be **VK_FORMAT_UNDEFINED**. Such conversions must only be used to sample image views with a matching **external format**. When creating an external format conversion, the value of **components** is ignored.
Valid Usage

- VUID-VkSamplerYcbcrConversionCreateInfo-format-01904
  If an external format conversion is being created, format must be VK_FORMAT_UNDEFINED

- VUID-VkSamplerYcbcrConversionCreateInfo-format-04061
  If an external format conversion is not being created, format must represent unsigned normalized values (i.e. the format must be a UNORM format)

- VUID-VkSamplerYcbcrConversionCreateInfo-format-01650
  The potential format features of the sampler Y’C_B conversion must support VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT or VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT

- VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01651
  If the potential format features of the sampler Y’C_B conversion do not support VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT, xChromaOffset and yChromaOffset must not be VK_CHROMA_LOCATION_COSITED_EVEN if the corresponding channels are downsampled

- VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-01652
  If the potential format features of the sampler Y’C_B conversion do not support VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT, xChromaOffset and yChromaOffset must not be VK_CHROMA_LOCATION_MIDPOINT if the corresponding channels are downsampled

- VUID-VkSamplerYcbcrConversionCreateInfo-components-02581
  If the format has a _422 or _420 suffix, then components.g must be the identity swizzle

- VUID-VkSamplerYcbcrConversionCreateInfo-components-02582
  If the format has a _422 or _420 suffix, then components.a must be the identity swizzle, VK_COMPONENT_SWIZZLE_ONE, or VK_COMPONENT_SWIZZLE_ZERO

- VUID-VkSamplerYcbcrConversionCreateInfo-components-02583
  If the format has a _422 or _420 suffix, then components.r must be the identity swizzle or VK_COMPONENT_SWIZZLE_B

- VUID-VkSamplerYcbcrConversionCreateInfo-components-02584
  If the format has a _422 or _420 suffix, then components.b must be the identity swizzle or VK_COMPONENT_SWIZZLE_R

- VUID-VkSamplerYcbcrConversionCreateInfo-components-02585
  If the format has a _422 or _420 suffix, and if either components.r or components.b is the identity swizzle, both values must be the identity swizzle

- VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrModel-01655
  If ycbcrModel is not VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY, then components.r, components.g, and components.b must correspond to channels of the format; that is, components.r, components.g, and components.b must not be VK_COMPONENT_SWIZZLE_ZERO or VK_COMPONENT_SWIZZLE_ONE, and must not correspond to a channel which contains zero or one as a consequence of conversion to RGBA

- VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrRange-02748
  If ycbcrRange is VK_SAMPLER_YCBCR_RANGE_ITU_NARROW then the R, G and B channels obtained by applying the component swizzle to format must each have a bit-depth greater than or equal to 8
If the potential format features of the sampler \( Y'C'_B'C'_R \) conversion do not support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCE_ABLE_BIT`, `forceExplicitReconstruction` must be `VK_FALSE`.

If the potential format features of the sampler \( Y'C'_B'C'_R \) conversion do not support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT`, `chromaFilter` must not be `VK_FILTER_LINEAR`.

**Valid Usage (Implicit)**

- **VUID-VkSamplerYcbcrConversionCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO`.

- **VUID-VkSamplerYcbcrConversionCreateInfo-pNext-pNext**
  
  `pNext` must be `NULL` or a pointer to a valid instance of `VkExternalFormatANDROID`.

  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkSamplerYcbcrConversionCreateInfo-format-parameter**
  
  `format` must be a valid `VkFormat` value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrModel-parameter**
  
  `ycbcrModel` must be a valid `VkSamplerYcbcrModelConversion` value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-ycbcrRange-parameter**
  
  `ycbcrRange` must be a valid `VkSamplerYcbcrRange` value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-components-parameter**
  
  `components` must be a valid `VkComponentMapping` structure.

- **VUID-VkSamplerYcbcrConversionCreateInfo-xChromaOffset-parameter**
  
  `xChromaOffset` must be a valid `VkChromaLocation` value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-yChromaOffset-parameter**
  
  `yChromaOffset` must be a valid `VkChromaLocation` value.

- **VUID-VkSamplerYcbcrConversionCreateInfo-chromaFilter-parameter**
  
  `chromaFilter` must be a valid `VkFilter` value.

If `chromaFilter` is `VK_FILTER_NEAREST`, chroma samples are reconstructed to luma channel resolution using nearest-neighbour sampling. Otherwise, chroma samples are reconstructed using interpolation. More details can be found in the description of sampler \( Y'C'_B'C'_R \) conversion in the Image Operations chapter.

`VkSamplerYcbcrModelConversion` defines the conversion from the source color model to the shader color model. Possible values are:

If `chromaFilter` is `VK_FILTER_NEAREST`, chroma samples are reconstructed to luma channel resolution using nearest-neighbour sampling. Otherwise, chroma samples are reconstructed using interpolation. More details can be found in the description of sampler \( Y'C'_B'C'_R \) conversion in the Image Operations chapter.

`VkSamplerYcbcrModelConversion` defines the conversion from the source color model to the shader color model. Possible values are:
or the equivalent

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkSamplerYcbcrModelConversion VkSamplerYcbcrModelConversionKHR;
```

- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY** specifies that the input values to the conversion are unmodified.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY** specifies no model conversion but the inputs are range expanded as for YCbCr.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709** specifies the color model conversion from Y'CbCr to R'G'B' defined in BT.709 and described in the “BT.709 Y'CbCr conversion” section of the Khronos Data Format Specification.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601** specifies the color model conversion from Y'CbCr to R'G'B' defined in BT.601 and described in the “BT.601 Y'CbCr conversion” section of the Khronos Data Format Specification.
- **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020** specifies the color model conversion from Y'CbCr to R'G'B' defined in BT.2020 and described in the “BT.2020 Y'CbCr conversion” section of the Khronos Data Format Specification.

In the **VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_*** color models, for the input to the sampler Y'CbCr range expansion and model conversion:
• the Y (Y’ luma) channel corresponds to the G channel of an RGB image.
• the CB (Cb or “U” blue color difference) channel corresponds to the B channel of an RGB image.
• the CR (Cr or “V” red color difference) channel corresponds to the R channel of an RGB image.
• the alpha channel, if present, is not modified by color model conversion.

These rules reflect the mapping of channels after the channel swizzle operation (controlled by `VkSamplerYcbcrConversionCreateInfo::components`).

**Note**
For example, an “YUVA” 32-bit format comprising four 8-bit channels can be implemented as `VK_FORMAT_R8G8B8A8_UNORM` with a component mapping:

- `components.a = VK_COMPONENT_SWIZZLE_IDENTITY`
- `components.r = VK_COMPONENT_SWIZZLE_B`
- `components.g = VK_COMPONENT_SWIZZLE_R`
- `components.b = VK_COMPONENT_SWIZZLE_G`

The `VkSamplerYcbcrRange` enum describes whether color channels are encoded using the full range of numerical values or whether values are reserved for headroom and foot room. `VkSamplerYcbcrRange` is defined as:

```c
// Provided by VK_VERSION_1_1
typedef enum VkSamplerYcbcrRange {
    VK_SAMPLER_YCBCR_RANGE_ITU_FULL = 0,
    VK_SAMPLER_YCBCR_RANGE_ITU_NARROW = 1,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_SAMPLER_YCBCR_RANGE_ITU_FULL_KHR = VK_SAMPLER_YCBCR_RANGE_ITU_FULL,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_SAMPLER_YCBCR_RANGE_ITU_NARROW_KHR = VK_SAMPLER_YCBCR_RANGE_ITU_NARROW,
} VkSamplerYcbcrRange;
```

or the equivalent

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkSamplerYcbcrRange VkSamplerYcbcrRangeKHR;
```

- `VK_SAMPLER_YCBCR_RANGE_ITU_FULL` specifies that the full range of the encoded values are valid and interpreted according to the ITU “full range” quantization rules.
- `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW` specifies that headroom and foot room are reserved in the numerical range of encoded values, and the remaining values are expanded according to the ITU “narrow range” quantization rules.

The formulae for these conversions is described in the [Sampler Y’CbCr Range Expansion](#) section of the [Image Operations](#) chapter.
No range modification takes place if `ycbcrModel` is `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`; the `ycbcrRange` field of `VkSamplerYcbcrConversionCreateInfo` is ignored in this case.

The `VkChromaLocation` enum defines the location of downscaled chroma channel samples relative to the luma samples, and is defined as:

```c
// Provided by VK_VERSION_1_1
typedef enum VkChromaLocation {
    VK_CHROMA_LOCATION_COSITED_EVEN = 0,
    VK_CHROMA_LOCATION_MIDPOINT = 1,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_CHROMA_LOCATION_COSITED_EVEN_KHR = VK_CHROMA_LOCATION_COSITED_EVEN,
    // Provided by VK_KHR_sampler_ycbcr_conversion
    VK_CHROMA_LOCATION_MIDPOINT_KHR = VK_CHROMA_LOCATION_MIDPOINT,
} VkChromaLocation;
```

or the equivalent

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkChromaLocation VkChromaLocationKHR;
```

- `VK_CHROMA_LOCATION_COSITED_EVEN` specifies that downscaled chroma samples are aligned with luma samples with even coordinates.
- `VK_CHROMA_LOCATION_MIDPOINT` specifies that downscaled chroma samples are located halfway between each even luma sample and the nearest higher odd luma sample.

To destroy a sampler `Y'C_bC_r` conversion, call:

```c
// Provided by VK_VERSION_1_1
void vkDestroySamplerYcbcrConversion(
    VkDevice device,
    VkSamplerYcbcrConversion ycbcrConversion,
    const VkAllocationCallbacks* pAllocator);
```

or the equivalent command

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
void vkDestroySamplerYcbcrConversionKHR(
    VkDevice device,
    VkSamplerYcbcrConversion ycbcrConversion,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the `Y'C_bC_r` conversion.
- `ycbcrConversion` is the conversion to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
Valid Usage (Implicit)

- VUID-vkDestroySamplerYcbcrConversion-device-parameter
device must be a valid VkDevice handle

- VUID-vkDestroySamplerYcbcrConversion-ycbcrConversion-parameter
  If ycbcrConversion is not VK_NULL_HANDLE, ycbcrConversion must be a valid VkSamplerYcbcrConversion handle

- VUID-vkDestroySamplerYcbcrConversion-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroySamplerYcbcrConversion-ycbcrConversion-parent
  If ycbcrConversion is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to ycbcrConversion must be externally synchronized

In addition to the predefined border color values, applications can provide a custom border color value by including the VkSamplerCustomBorderColorCreateInfoEXT structure in the VkSamplerCreateInfo::pNext chain.

The VkSamplerCustomBorderColorCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_custom_border_color
typedef struct VkSamplerCustomBorderColorCreateInfoEXT {
    VkStructureType      sType;
    const void*          pNext;
    VkClearColorValue    customBorderColor;
    VkFormat             format;
} VkSamplerCustomBorderColorCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `customBorderColor` is a VkClearColorValue representing the desired custom sampler border color.
- `format` is a VkFormat representing the format of the sampled image view(s). This field may be VK_FORMAT_UNDEFINED if the customBorderColorWithoutFormat feature is enabled.
Valid Usage

- VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-04013
  If provided \texttt{format} is not \texttt{VK\_FORMAT\_UNDEFINED} then the \texttt{VkSamplerCreateInfo:borderColor} type \textbf{must} match the sampled type of the provided \texttt{format}, as shown in the \textit{SPIR-V Sampled Type} column of the \textbf{Interpretation of Numeric Format} table.

- VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-04014
  If the \texttt{customBorderColorWithoutFormat} feature is not enabled then \texttt{format} \textbf{must} not be \texttt{VK\_FORMAT\_UNDEFINED}.

- VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-04015
  If the sampler is used to sample an image view of \texttt{VK\_FORMAT\_B4G4R4A4\_UNORM\_PACK16}, \texttt{VK\_FORMAT\_B5G6R5\_UNORM\_PACK16}, or \texttt{VK\_FORMAT\_B5G5R5A1\_UNORM\_PACK16} format then \texttt{format} \textbf{must} not be \texttt{VK\_FORMAT\_UNDEFINED}.

Valid Usage (Implicit)

- VUID-VkSamplerCustomBorderColorCreateInfoEXT-sType-sType
  \texttt{sType} \textbf{must} be \texttt{VK\_STRUCTURE\_TYPE\_SAMPLER\_CUSTOM\_BORDER\_COLOR\_CREATE\_INFO\_EXT}.

- VUID-VkSamplerCustomBorderColorCreateInfoEXT-format-parameter
  \texttt{format} \textbf{must} be a valid \texttt{VkFormat} value.
Chapter 14. Resource Descriptors

A descriptor is an opaque data structure representing a shader resource such as a buffer, buffer view, image view, sampler, or combined image sampler. Descriptors are organised into descriptor sets, which are bound during command recording for use in subsequent draw commands. The arrangement of content in each descriptor set is determined by a descriptor set layout, which determines what descriptors can be stored within it. The sequence of descriptor set layouts that can be used by a pipeline is specified in a pipeline layout. Each pipeline object can use up to maxBoundDescriptorSets (see Limits) descriptor sets.

Shaders access resources via variables decorated with a descriptor set and binding number that link them to a descriptor in a descriptor set. The shader interface mapping to bound descriptor sets is described in the Shader Resource Interface section.

Shaders can also access buffers without going through descriptors by using Physical Storage Buffer Access to access them through 64-bit addresses.

14.1. Descriptor Types

There are a number of different types of descriptor supported by Vulkan, corresponding to different resources or usage. The following sections describe the API definitions of each descriptor type. The mapping of each type to SPIR-V is listed in the Shader Resource and Descriptor Type Correspondence and Shader Resource and Storage Class Correspondence tables in the Shader Interfaces chapter.

14.1.1. Storage Image

A storage image (VK_DESCRIPTOR_TYPE_STORAGE_IMAGE) is a descriptor type associated with an image resource via an image view that load, store, and atomic operations can be performed on.

Storage image loads are supported in all shader stages for image views whose format features contain VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT.

Stores to storage images are supported in compute shaders for image views whose format features contain VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT.

Atomic operations on storage images are supported in compute shaders for image views whose format features contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT.

When the fragmentStoresAndAtomics feature is enabled, stores and atomic operations are also supported for storage images in fragment shaders with the same set of image formats as supported in compute shaders. When the vertexPipelineStoresAndAtomics feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of image formats as supported in compute shaders.

The image subresources for a storage image must be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR or VK_IMAGE_LAYOUT_GENERAL layout in order to access its data in a shader.
14.1.2. Sampler

A sampler descriptor (VK_DESCRIPTOR_TYPE_SAMPLER) is a descriptor type associated with a sampler object, used to control the behavior of sampling operations performed on a sampled image.

14.1.3. Sampled Image

A sampled image (VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE) is a descriptor type associated with an image resource via an image view that sampling operations can be performed on.

Shaders combine a sampled image variable and a sampler variable to perform sampling operations.

Sampled images are supported in all shader stages for image views whose format features contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT.

The image subresources for a sampled image must be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout in order to access its data in a shader.

14.1.4. Combined Image Sampler

A combined image sampler (VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER) is a single descriptor type associated with both a sampler and an image resource, combining both a sampler and sampled image descriptor into a single descriptor.

If the descriptor refers to a sampler that performs Y’C_bC_r conversion or samples a subsampled image, the sampler must only be used to sample the image in the same descriptor. Otherwise, the sampler and image in this type of descriptor can be used freely with any other samplers and images.

The image subresources for a combined image sampler must be in the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL, or VK_IMAGE_LAYOUT_GENERAL layout in order to access its data in a shader.

**Note**

On some implementations, it may be more efficient to sample from an image using a combination of sampler and sampled image that are stored together in the descriptor set in a combined descriptor.

14.1.5. Uniform Texel Buffer

A uniform texel buffer (VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER) is a descriptor type associated with a buffer resource via a buffer view that formatted load operations can be performed on.
Uniform texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image.

Load operations from uniform texel buffers are supported in all shader stages for image formats which report support for the `VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

### 14.1.6. Storage Texel Buffer

A *storage texel buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`) is a descriptor type associated with a *buffer resource* via a *buffer view* that formatted load, store, and atomic operations *can* be performed on.

Storage texel buffers define a tightly-packed 1-dimensional linear array of texels, with texels going through format conversion when read in a shader in the same way as they are for an image. Unlike *uniform texel buffers*, these buffers can also be written to in the same way as for *storage images*.

Storage texel buffer loads are supported in all shader stages for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

Stores to storage texel buffers are supported in compute shaders for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

Atomic operations on storage texel buffers are supported in compute shaders for texel buffer formats which report support for the `VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT` feature bit via `vkGetPhysicalDeviceFormatProperties` in `VkFormatProperties::bufferFeatures`.

When the `fragmentStoresAndAtomics` feature is enabled, stores and atomic operations are also supported for storage texel buffers in fragment shaders with the same set of texel buffer formats as supported in compute shaders. When the `vertexPipelineStoresAndAtomics` feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of texel buffer formats as supported in compute shaders.

### 14.1.7. Storage Buffer

A *storage buffer* (`VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`) is a descriptor type associated with a *buffer resource* directly, described in a shader as a structure with various members that load, store, and atomic operations *can* be performed on.

Note

Atomic operations *can* only be performed on members of certain types as defined in the SPIR-V environment appendix.

### 14.1.8. Uniform Buffer

A *uniform buffer* (`VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`) is a descriptor type associated with a *buffer resource* directly, described in a shader as a structure with various members that load operations
can be performed on.

### 14.1.9. Dynamic Uniform Buffer

A *dynamic uniform buffer* (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC) is almost identical to a *uniform buffer*, and differs only in how the offset into the buffer is specified. The base offset calculated by the VkDescriptorBufferInfo when initially updating the descriptor set is added to a *dynamic offset* when binding the descriptor set.

### 14.1.10. Dynamic Storage Buffer

A *dynamic storage buffer* (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC) is almost identical to a *storage buffer*, and differs only in how the offset into the buffer is specified. The base offset calculated by the VkDescriptorBufferInfo when initially updating the descriptor set is added to a *dynamic offset* when binding the descriptor set.

### 14.1.11. Inline Uniform Block

An *inline uniform block* (VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT) is almost identical to a *uniform buffer*, and differs only in taking its storage directly from the encompassing descriptor set instead of being backed by buffer memory. It is typically used to access a small set of constant data that does not require the additional flexibility provided by the indirection enabled when using a uniform buffer where the descriptor and the referenced buffer memory are decoupled. Compared to push constants, they allow reusing the same set of constant data across multiple disjoint sets of draw and dispatch commands.

Inline uniform block descriptors *cannot* be aggregated into arrays. Instead, the array size specified for an inline uniform block descriptor binding specifies the binding’s capacity in bytes.

### 14.1.12. Input Attachment

An *input attachment* (VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT) is a descriptor type associated with an *image resource* via an *image view* that can be used for framebuffer local load operations in fragment shaders.

All image formats that are supported for color attachments (VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT) or depth/stencil attachments (VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT) for a given image tiling mode are also supported for input attachments.

The image subresources for an input attachment *must* be in a *valid image layout* in order to access its data in a shader.

### 14.1.13. Acceleration Structure

An *acceleration structure* (VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR or VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV) is a descriptor type that is used to retrieve scene geometry from within shaders that are used for ray traversal. Shaders have read-only access to the memory.
14.1.14. Mutable

A descriptor of mutable (VK_DESCRIPTOR_TYPE_MUTABLE_VALVE) type indicates that this descriptor can mutate to any of the descriptor types given in the VkMutableDescriptorTypeCreateInfoVALVE::pDescriptorTypes list of descriptor types in the pNext chain of VkDescriptorSetLayoutCreateInfo for this binding. At any point, each individual descriptor of mutable type has an active descriptor type. The active descriptor type can be any one of the declared types in pDescriptorTypes. Additionally, a mutable descriptor’s active descriptor type can be of the VK_DESCRIPTOR_TYPE_MUTABLE_VALVE type, which is the initial active descriptor type. The active descriptor type can change when the descriptor is updated. When a descriptor is consumed by binding a descriptor set, the active descriptor type is considered, not VK_DESCRIPTOR_TYPE_MUTABLE_VALVE.

An active descriptor type of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE is considered an undefined descriptor. If a descriptor is consumed where the active descriptor type does not match what the shader expects, the descriptor is considered an undefined descriptor.

Note
To find which descriptor types are supported as VK_DESCRIPTOR_TYPE_MUTABLE_VALVE, the application can use vkGetDescriptorSetLayoutSupport with an VK_DESCRIPTOR_TYPE_MUTABLE_VALVE binding, with the list of descriptor types to query in the VkMutableDescriptorTypeCreateInfoVALVE::pDescriptorTypes array for that binding.

Note
The intention of a mutable descriptor type is that implementations allocate N bytes per descriptor, where N is determined by the maximum descriptor size for a given descriptor binding. Implementations are not expected to keep track of the active descriptor type, and it should be considered a C-like union type.

A mutable descriptor type is not considered as efficient in terms of run-time performance as using a non-mutable descriptor type, and applications are not encouraged to use them outside API layering efforts. Mutable descriptor types can be more efficient if the alternative is using many different descriptors to emulate mutable descriptor types.

14.2. Descriptor Sets

Descriptors are grouped together into descriptor set objects. A descriptor set object is an opaque object containing storage for a set of descriptors, where the types and number of descriptors is defined by a descriptor set layout. The layout object may be used to define the association of each descriptor binding with memory or other implementation resources. The layout is used both for determining the resources that need to be associated with the descriptor set, and determining the interface between shader stages and shader resources.
14.2.1. Descriptor Set Layout

A descriptor set layout object is defined by an array of zero or more descriptor bindings. Each individual descriptor binding is specified by a descriptor type, a count (array size) of the number of descriptors in the binding, a set of shader stages that can access the binding, and (if using immutable samplers) an array of sampler descriptors.

Descriptor set layout objects are represented by `VkDescriptorSetLayout` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSetLayout)
```

To create descriptor set layout objects, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateDescriptorSetLayout(
    VkDevice device,                  // Provided by VK_VERSION_1_0
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,   // Provided by VK_VERSION_1_0
    const VkAllocationCallbacks* pAllocator,      // Provided by VK_VERSION_1_0
    VkDescriptorSetLayout* pSetLayout);         // Provided by VK_VERSION_1_0
```

- `device` is the logical device that creates the descriptor set layout.
- `pCreateInfo` is a pointer to a `VkDescriptorSetLayoutCreateInfo` structure specifying the state of the descriptor set layout object.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pSetLayout` is a pointer to a `VkDescriptorSetLayout` handle in which the resulting descriptor set layout object is returned.

**Valid Usage (Implicit)**

- VUID-vkCreateDescriptorSetLayout-device-parameter
  ```
  device must be a valid VkDevice handle
  ```

- VUID-vkCreateDescriptorSetLayout-pCreateInfo-parameter
  ```
  pCreateInfo must be a valid pointer to a valid VkDescriptorSetLayoutCreateInfo structure
  ```

- VUID-vkCreateDescriptorSetLayout-pAllocator-parameter
  ```
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
  ```

- VUID-vkCreateDescriptorSetLayout-pSetLayout-parameter
  ```
  pSetLayout must be a valid pointer to a VkDescriptorSetLayout handle
  ```
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

Information about the descriptor set layout is passed in a VkDescriptorSetLayoutCreateInfo structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType        sType;
    const void*             pNext;
    VkDescriptorSetLayoutCreateFlags flags;
    uint32_t                bindingCount;
    const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is a bitmask of VkDescriptorSetLayoutCreateFlagBits specifying options for descriptor set layout creation.
- bindingCount is the number of elements in pBindings.
- pBindings is a pointer to an array of VkDescriptorSetLayoutBinding structures.
Valid Usage

- VUID-VkDescriptorSetLayoutCreateInfo-binding-00279
  The VkDescriptorSetLayoutBinding::binding members of the elements of the pBindings array must each have different values.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-00280
  If flags contains VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR, then all elements of pBindings must not have a descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-02208
  If flags contains VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR, then all elements of pBindings must not have a descriptorType of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-00281
  If flags contains VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR, then the total number of elements of all bindings must be less than or equal to VkPhysicalDevicePushDescriptorPropertiesKHR::maxPushDescriptors.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-04590
  If flags contains VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR, flags must not contain VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-04591
  If flags contains VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR, pBindings must not have a descriptorType of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-03000
  If any binding has the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT bit set, flags must include VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT.

- VUID-VkDescriptorSetLayoutCreateInfo-descriptorType-03001
  If any binding has the VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT bit set, then all bindings must not have descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-04592
  If flags contains VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT, flags must not contain VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE.

- VUID-VkDescriptorSetLayoutCreateInfo-descriptorType-04593
  If any binding has a descriptorType value of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE, then a VkMutableDescriptorTypeCreateInfoVALVE must be present in the pNext chain.

- VUID-VkDescriptorSetLayoutCreateInfo-descriptorType-04594
  If a binding has a descriptorType value of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE, then pImmutableSamplers must be NULL.

- VUID-VkDescriptorSetLayoutCreateInfo-flags-04595
  If VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE::mutableDescriptorType is not enabled, pBindings must not contain a descriptorType of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE.
If `flags` contains `VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE`, `VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE::mutableDescriptorType` must be enabled.

### Valid Usage (Implicit)

- **VUID-VkDescriptorSetLayoutCreateInfo-sType-sType**
  
  - `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO`

- **VUID-VkDescriptorSetLayoutCreateInfo-pNext-pNext**

  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDescriptorSetLayoutBindingFlagsCreateInfo` or `VkMutableDescriptorTypeCreateInfoVALVE`

- **VUID-VkDescriptorSetLayoutCreateInfo-sType-unique**

  The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkDescriptorSetLayoutCreateInfo-flags-parameter**

  `flags` must be a valid combination of `VkDescriptorSetLayoutCreateFlagBits` values

- **VUID-VkDescriptorSetLayoutCreateInfo-pBindings-parameter**

  If `bindingCount` is not 0, `pBindings` must be a valid pointer to an array of `bindingCount` valid `VkDescriptorSetLayoutBinding` structures

Information about the possible descriptor types for mutable descriptor types is passed in a `VkMutableDescriptorTypeCreateInfoVALVE` structure as a `pNext` to a `VkDescriptorSetLayoutCreateInfo` structure or a `VkDescriptorPoolCreateInfo` structure.

The `VkMutableDescriptorTypeCreateInfoVALVE` structure is defined as:

```c
// Provided by VK_VALVE_mutable_descriptor_type
typedef struct VkMutableDescriptorTypeCreateInfoVALVE {
  VkStructureType                            sType;
  const void*                                pNext;
  uint32_t                                   mutableDescriptorTypeListCount;
  const VkMutableDescriptorTypeListVALVE*    pMutableDescriptorTypeLists;
} VkMutableDescriptorTypeCreateInfoVALVE;
```

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to a structure extending this structure.

- `mutableDescriptorTypeListCount` is the number of elements in `pMutableDescriptorTypeLists`.

- `pMutableDescriptorTypeLists` is a pointer to an array of `VkMutableDescriptorTypeListVALVE` structures.

If `mutableDescriptorTypeListCount` is zero or if this structure is not included in the `pNext` chain, the `VkMutableDescriptorTypeListVALVE` for each element is considered to be zero or `NULL` for each member. Otherwise, the descriptor set layout binding at `VkDescriptorSetLayoutCreateInfo`
::pBindings[i] uses the descriptor type lists in VkMutableDescriptorTypeCreateInfoVALVE::pMutableDescriptorTypeLists[i].

Valid Usage (Implicit)

- VUID-VkMutableDescriptorTypeCreateInfoVALVE-sType-sType
  sType must be VK_STRUCTURE_TYPE_MUTABLE_DESCRIPTOR_TYPE_CREATE_INFO_VALVE

- VUID-VkMutableDescriptorTypeCreateInfoVALVE-pMutableDescriptorTypeLists-parameter
  If mutableDescriptorTypeListCount is not 0, pMutableDescriptorTypeLists must be a valid pointer to an array of mutableDescriptorTypeListCount valid VkMutableDescriptorTypeListVALVE structures

The list of potential descriptor types a given mutable descriptor can mutate to is passed in a VkMutableDescriptorTypeListVALVE structure.

The VkMutableDescriptorTypeListVALVE structure is defined as:

```c
// Provided by VK_VALVE_mutable_descriptor_type
typedef struct VkMutableDescriptorTypeListVALVE {
    uint32_t descriptorTypeCount;
    const VkDescriptorType* pDescriptorTypes;
} VkMutableDescriptorTypeListVALVE;
```

- descriptorTypeCount is the number of elements in pDescriptorTypes.
- pDescriptorTypes is NULL or a pointer to an array of descriptorTypeCount VkDescriptorType values which define which descriptor types a given binding may mutate to.
Valid Usage

- VUID-VkMutableDescriptorTypeListVALVE-descriptorTypeCount-04597
  `descriptorTypeCount` must not be 0 if the corresponding binding is of `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`

- VUID-VkMutableDescriptorTypeListVALVE-pDescriptorTypes-04598
  `pDescriptorTypes` must be a valid pointer to an array of `descriptorTypeCount` valid, unique `VkDescriptorType` values if the given binding is of `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE` type

- VUID-VkMutableDescriptorTypeListVALVE-descriptorTypeCount-04599
  `descriptorTypeCount` must be 0 if the corresponding binding is not of `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`

- VUID-VkMutableDescriptorTypeListVALVE-pDescriptorTypes-04600
  `pDescriptorTypes` must not contain `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`

- VUID-VkMutableDescriptorTypeListVALVE-pDescriptorTypes-04601
  `pDescriptorTypes` must not contain `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`

- VUID-VkMutableDescriptorTypeListVALVE-pDescriptorTypes-04602
  `pDescriptorTypes` must not contain `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`

- VUID-VkMutableDescriptorTypeListVALVE-pDescriptorTypes-04603
  `pDescriptorTypes` must not contain `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`

Valid Usage (Implicit)

- VUID-VkMutableDescriptorTypeListVALVE-pDescriptorTypes-parameter
  If `descriptorTypeCount` is not 0, `pDescriptorTypes` must be a valid pointer to an array of `descriptorTypeCount` valid `VkDescriptorType` values

Bits which can be set in `VkDescriptorSetLayoutCreateInfo::flags` to specify options for descriptor set layout are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorSetLayoutCreateFlagBits {
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT = 0x00000002,
    // Provided by VK_KHR_push_descriptor
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR = 0x00000001,
    // Provided by VK_VALVE_mutable_descriptor_type
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE = 0x00000004,
    // Provided by VK_EXT_descriptor_indexing
    VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT = VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT,
} VkDescriptorSetLayoutCreateFlagBits;
```

- `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR` specifies that descriptor sets must not be allocated using this layout, and descriptors are instead pushed by
vkCmdPushDescriptorSetKHR.

- **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** specifies that descriptor sets using this layout must be allocated from a descriptor pool created with the **VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT** bit set. Descriptor set layouts created with this bit set have alternate limits for the maximum number of descriptors per-stage and per-pipeline layout. The non-UpdateAfterBind limits only count descriptors in sets created without this flag. The UpdateAfterBind limits count all descriptors, but the limits may be higher than the non-UpdateAfterBind limits.

- **VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT** specifies that descriptor sets using this layout must be allocated from a descriptor pool created with the **VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT** bit set. Descriptor set layouts created with this bit have no expressable limit for maximum number of descriptors per-stage. Host descriptor sets are limited only by available host memory, but may be limited for implementation specific reasons. Implementations may limit the number of supported descriptors to UpdateAfterBind limits or non-UpdateAfterBind limits, whichever is larger.

```
// Provided by VK_VERSION_1_0
typedef VkFlags VkDescriptorSetLayoutCreateFlags;
```

**VkDescriptorSetLayoutCreateFlags** is a bitmask type for setting a mask of zero or more **VkDescriptorSetLayoutCreateFlagBits**.

The **VkDescriptorSetLayoutBinding** structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t              binding;
    VkDescriptorType      descriptorType;
    uint32_t              descriptorCount;
    VkShaderStageFlags    stageFlags;
    const VkSampler*      pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

- **binding** is the binding number of this entry and corresponds to a resource of the same binding number in the shader stages.

- **descriptorType** is a **VkDescriptorType** specifying which type of resource descriptors are used for this binding.

- **descriptorCount** is the number of descriptors contained in the binding, accessed in a shader as an array, except if **descriptorType** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT** in which case **descriptorCount** is the size in bytes of the inline uniform block. If **descriptorCount** is zero this binding entry is reserved and the resource must not be accessed from any stage via this binding within any pipeline using the set layout.

- **stageFlags** member is a bitmask of **VkShaderStageFlagBits** specifying which pipeline shader stages can access a resource for this binding. **VK_SHADER_STAGE_ALL** is a shorthand specifying that all defined shader stages, including any additional stages defined by extensions, can access the
If a shader stage is not included in `stageFlags`, then a resource **must** not be accessed from that stage via this binding within any pipeline using the set layout. Other than input attachments which are limited to the fragment shader, there are no limitations on what combinations of stages **can** use a descriptor binding, and in particular a binding **can** be used by both graphics stages and the compute stage.

- `pImmutableSamplers` affects initialization of samplers. If `descriptorType` specifies a `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` type descriptor, then `pImmutableSamplers` **can** be used to initialize a set of *immutable samplers*. Immutable samplers are permanently bound into the set layout and **must** not be changed; updating a `VK_DESCRIPTOR_TYPE_SAMPLER` descriptor with immutable samplers is not allowed and updates to a `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` descriptor with immutable samplers does not modify the samplers (the image views are updated, but the sampler updates are ignored). If `pImmutableSamplers` is not `NULL`, then it points to an array of sampler handles that will be copied into the set layout and used for the corresponding binding. Only the sampler handles **must** not be destroyed before the final use of the set layout and any descriptor pools and sets created using it. If `pImmutableSamplers` is `NULL`, then the sampler slots are dynamic and sampler handles **must** be bound into descriptor sets using this layout. If `descriptorType` is not one of these descriptor types, then `pImmutableSamplers` is **ignored**.

The above layout definition allows the descriptor bindings to be specified sparsely such that not all binding numbers between 0 and the maximum binding number need to be specified in the `pBindings` array. Bindings that are not specified have a `descriptorCount` and `stageFlags` of zero, and the value of `descriptorType` is undefined. However, all binding numbers between 0 and the maximum binding number in the `VkDescriptorSetLayoutCreateInfo::pBindings` array **may** consume memory in the descriptor set layout even if not all descriptor bindings are used, though it **should** not consume additional memory from the descriptor pool.

---

**Note**

The maximum binding number specified **should** be as compact as possible to avoid wasted memory.
Valid Usage

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-00282**
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `descriptorCount` is not 0 and `pImmutableSamplers` is not NULL, `pImmutableSamplers` must be a valid pointer to an array of `descriptorCount` valid `VkSampler` handles.

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-04604**
  If the `inlineUniformBlock` feature is not enabled, `descriptorType` must not be `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`.

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-02209**
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` must be a multiple of 4.

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-02210**
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` must be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxInlineUniformBlockSize`.

- **VUID-VkDescriptorSetLayoutBinding-descriptorCount-00283**
  If `descriptorCount` is not 0, `stageFlags` must be a valid combination of `VkShaderStageFlagBits` values.

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-01510**
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` and `descriptorCount` is not 0, then `stageFlags` must be 0 or `VK_SHADER_STAGE_FRAGMENT_BIT`.

- **VUID-VkDescriptorSetLayoutBinding-pImmuneSamplers-04009**
  The sampler objects indicated by `pImmutableSamplers` must not have a `borderColor` with one of the values `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT`.

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-04605**
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`, then `pImmutableSamplers` must be NULL.

Valid Usage (Implicit)

- **VUID-VkDescriptorSetLayoutBinding-descriptorType-parameter**
  `descriptorType` must be a valid `VkDescriptorType` value.

If the `pNext` chain of a `VkDescriptorSetLayoutCreateInfo` structure includes a `VkDescriptorSetLayoutBindingFlagsCreateInfo` structure, then that structure includes an array of flags, one for each descriptor set layout binding.

The `VkDescriptorSetLayoutBindingFlagsCreateInfo` structure is defined as:
typedef struct VkDescriptorSetLayoutBindingFlagsCreateInfo {
    VkStructureType                      sType;
    const void*                          pNext;
    uint32_t                              bindingCount;
    const VkDescriptorBindingFlags*      pBindingFlags;
} VkDescriptorSetLayoutBindingFlagsCreateInfo;

or the equivalent

// Provided by VK_EXT_descriptor_indexing
typedef VkDescriptorSetLayoutBindingFlagsCreateInfo VkDescriptorSetLayoutBindingFlagsCreateInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **bindingCount** is zero or the number of elements in **pBindingFlags**.
- **pBindingFlags** is a pointer to an array of **VkDescriptorBindingFlags** bitfields, one for each descriptor set layout binding.

If **bindingCount** is zero or if this structure is not included in the **pNext** chain, the **VkDescriptorBindingFlags** for each descriptor set layout binding is considered to be zero. Otherwise, the descriptor set layout binding at **VkDescriptorSetLayoutCreateInfo::pBindings[i]** uses the flags in **pBindingFlags[i]**.
Valid Usage

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-bindingCount-03002
  If `bindingCount` is not zero, `bindingCount` must equal `VkDescriptorSetLayoutCreateInfo::bindingCount`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-flags-03003
  If `VkDescriptorSetLayoutCreateInfo::flags` includes `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`, then all elements of `pBindingFlags` must not include `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`, `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT`, or `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-pBindingFlags-03004
  If an element of `pBindingFlags` includes `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT`, then all other elements of `VkDescriptorSetLayoutCreateInfo::pBindings` must have a smaller value of `binding`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingUniformBufferUpdateAfterBind-03005
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUniformBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingSampledImageUpdateAfterBind-03006
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingSampledImageUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageImageUpdateAfterBind-03007
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingStorageImageUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageBufferUpdateAfterBind-03008
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingStorageBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingUniformTexelBufferUpdateAfterBind-03009
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUniformTexelBufferUpdateAfterBind` is not enabled, all bindings with descriptor type `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

- VUID-VkDescriptorSetLayoutBindingFlagsCreateInfo-descriptorBindingStorageTexelBufferUpdateAfterBind-03010
  If `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingStorageTexelBufferUpdateAfterBind` is not enabled, all bindings with...
When binding descriptors to a set layout, certain descriptor types must not use the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`. These types include:

- **VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER**
  - Must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` if
    - `VkPhysicalDeviceInlineUniformBlockFeaturesEXT::descriptorBindingInlineUniformBlockUpdateAfterBind` is not enabled.
  - All bindings with descriptor type **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT** must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` if
    - `VkPhysicalDeviceInlineUniformBlockFeaturesEXT::descriptorBindingInlineUniformBlockUpdateAfterBind` is not enabled.

- **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT**
  - Must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` if
    - `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUpdateUnusedWhilePending` is not enabled.

- **VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC**
  - Must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` if
    - `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUpdateUnusedWhilePending` is not enabled.

- **VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC**
  - Must not use `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` if
    - `VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingUpdateUnusedWhilePending` is not enabled.

Valid Usage (Implicit):

- **sType** must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO`.
- **pBindingFlags** must be a valid pointer to an array of `bindingCount` valid combinations of `VkDescriptorBindingFlagBits` values if `bindingCount` is not 0.
Bits which **can** be set in each element of `VkDescriptorSetLayoutBindingFlagsCreateInfo::pBindingFlags` to specify options for the corresponding descriptor set layout binding are:

```cpp
typedef enum VkDescriptorBindingFlagBits {
    VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT = 0x00000001,
    VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT = 0x00000002,
    VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT = 0x00000004,
    VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT = 0x00000008,
    // Provided by VK_EXT_descriptor_indexing
    VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT = VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT,
    // Provided by VK_EXT_descriptor_indexing
    VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT = VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT,
    // Provided by VK_EXT_descriptor_indexing
    VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT = VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT,
    // Provided by VK_EXT_descriptor_indexing
    VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT = VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT,
} VkDescriptorBindingFlagBits;
```

or the equivalent

```cpp
// Provided by VK_EXT_descriptor_indexing
typedef VkDescriptorBindingFlagBits VkDescriptorBindingFlagBitsEXT;
```

- **VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT** indicates that if descriptors in this binding are updated between when the descriptor set is bound in a command buffer and when that command buffer is submitted to a queue, then the submission will use the most recently set descriptors for this binding and the updates do not invalidate the command buffer. Descriptor bindings created with this flag are also partially exempt from the external synchronization requirement in `vkUpdateDescriptorSetWithTemplateKHR` and `vkUpdateDescriptorSets`. Multiple descriptors with this flag set **can** be updated concurrently in different threads, though the same descriptor **must** not be updated concurrently by two threads. Descriptors with this flag set **can** be updated concurrently with the set being bound to a command buffer in another thread, but not concurrently with the set being reset or freed.

- **VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT** indicates that descriptors in this binding that are not **dynamically used** need not contain valid descriptors at the time the descriptors are consumed. A descriptor is dynamically used if any shader invocation executes an instruction that performs any memory access using the descriptor.

- **VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT** indicates that descriptors in this binding **can** be updated after a command buffer has bound this descriptor set, or while a command buffer that uses this descriptor set is pending execution, as long as the descriptors that are updated are not used by those command buffers. If **VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT** is also set, then descriptors **can** be updated as long
as they are not dynamically used by any shader invocations. If
\texttt{VK\_DESCRIPTOR\_BINDING\_PARTIALLY\_BOUND\_BIT} is not set, then descriptors \textbf{can} be updated as long as
they are not statically used by any shader invocations.

- \texttt{VK\_DESCRIPTOR\_BINDING\_VARIABLE\_DESCRIPTOR\_COUNT\_BIT} indicates that this descriptor binding has
a variable size that will be specified when a descriptor set is allocated using this layout. The
value of \texttt{descriptorCount} is treated as an upper bound on the size of the binding. This \textbf{must} only
be used for the last binding in the descriptor set layout (i.e. the binding with the largest value of
binding). For the purposes of counting against limits such as \texttt{maxDescriptorSet*} and
\texttt{maxPerStageDescriptor*}, the full value of \texttt{descriptorCount} is counted, except for descriptor
bindings with a descriptor type of \texttt{VK\_DESCRIPTOR\_TYPE\_INLINE\_UNIFORM\_BLOCK\_EXT} where
\texttt{descriptorCount} specifies the upper bound on the byte size of the binding, thus it counts against
the \texttt{maxInlineUniformBlockSize} limit instead.

\textbf{Note}

Note that while \texttt{VK\_DESCRIPTOR\_BINDING\_UPDATE\_AFTER\_BIND\_BIT} and
\texttt{VK\_DESCRIPTOR\_BINDING\_UPDATE\_UNUSED\_WHILE\_PENDING\_BIT} both involve updates to
descriptor sets after they are bound, \texttt{VK\_DESCRIPTOR\_BINDING\_UPDATE\_UNUSED\_WHILE\_PENDING\_BIT} is a weaker requirement
since it is only about descriptors that are not used, whereas \texttt{VK\_DESCRIPTOR\_BINDING\_UPDATE\_AFTER\_BIND\_BIT} requires the implementation to
observe updates to descriptors that are used.

typedef VkFlags VkDescriptorBindingFlags;

or the equivalent

```c
// Provided by VK\_EXT\_descriptor\_indexing
typedef VkDescriptorBindingFlags VkDescriptorBindingFlagsEXT;
```

\texttt{VkDescriptorBindingFlags} is a bitmask type for setting a mask of zero or more
\texttt{VkDescriptorBindingFlagBits}.

To query information about whether a descriptor set layout \textbf{can} be created, call:

```c
// Provided by VK\_VERSION\_1\_1
void vkGetDescriptorSetLayoutSupport(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    VkDescriptorSetLayoutSupport* pSupport);
```

or the equivalent command
void vkGetDescriptorSetLayoutSupportKHR(
    VkDevice device,
    const VkDescriptorSetLayoutCreateInfo* pCreateInfo,
    VkDescriptorSetLayoutSupport* pSupport);

- `device` is the logical device that would create the descriptor set layout.
- `pCreateInfo` is a pointer to a `VkDescriptorSetLayoutCreateInfo` structure specifying the state of the descriptor set layout object.
- `pSupport` is a pointer to a `VkDescriptorSetLayoutSupport` structure, in which information about support for the descriptor set layout object is returned.

Some implementations have limitations on what fits in a descriptor set which are not easily expressible in terms of existing limits like `maxDescriptorSet*`, for example if all descriptor types share a limited space in memory but each descriptor is a different size or alignment. This command returns information about whether a descriptor set satisfies this limit. If the descriptor set layout satisfies the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, this command is guaranteed to return `VK_TRUE` in `VkDescriptorSetLayoutSupport::supported`. If the descriptor set layout exceeds the `VkPhysicalDeviceMaintenance3Properties::maxPerSetDescriptors` limit, whether the descriptor set layout is supported is implementation-dependent and `may` depend on whether the descriptor sizes and alignments cause the layout to exceed an internal limit.

This command does not consider other limits such as `maxPerStageDescriptor*`, and so a descriptor set layout that is supported according to this command `must` still satisfy the pipeline layout limits such as `maxPerStageDescriptor*` in order to be used in a pipeline layout.

**Note**

This is a `VkDevice` query rather than `VkPhysicalDevice` because the answer `may` depend on enabled features.

**Valid Usage (Implicit)**

- `VUID-vkGetDescriptorSetLayoutSupport-device-parameter`
  
  `device` must be a valid `VkDevice` handle

- `VUID-vkGetDescriptorSetLayoutSupport-pCreateInfo-parameter`
  
  `pCreateInfo` must be a valid pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure

- `VUID-vkGetDescriptorSetLayoutSupport-pSupport-parameter`
  
  `pSupport` must be a valid pointer to a `VkDescriptorSetLayoutSupport` structure

Information about support for the descriptor set layout is returned in a `VkDescriptorSetLayoutSupport` structure:
typedef struct VkDescriptorSetLayoutSupport {
    VkStructureType          sType;
    void*                     pNext;
    VkBool32                  supported;
} VkDescriptorSetLayoutSupport;

or the equivalent

typedef VkDescriptorSetLayoutSupport VkDescriptorSetLayoutSupportKHR;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `supported` specifies whether the descriptor set layout can be created.

`supported` is set to `VK_TRUE` if the descriptor set can be created, or else is set to `VK_FALSE`.

Valid Usage (Implicit)

- VUID-VkDescriptorSetLayoutSupport-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT`

- VUID-VkDescriptorSetLayoutSupport-pNext-pNext
  `pNext` must be `NULL` or a pointer to a valid instance of `VkDescriptorSetVariableDescriptorCountLayoutSupport`

- VUID-VkDescriptorSetLayoutSupport-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique

If the `pNext` chain of a `VkDescriptorSetLayoutSupport` structure includes a `VkDescriptorSetVariableDescriptorCountLayoutSupport` structure, then that structure returns additional information about whether the descriptor set layout is supported.

typedef struct VkDescriptorSetVariableDescriptorCountLayoutSupport {
    VkStructureType          sType;
    void*                     pNext;
    uint32_t                  maxVariableDescriptorCount;
} VkDescriptorSetVariableDescriptorCountLayoutSupport;

or the equivalent
typedef VkDescriptorSetVariableDescriptorCountLayoutSupport
VkDescriptorSetVariableDescriptorCountLayoutSupportEXT;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• maxVariableDescriptorCount indicates the maximum number of descriptors supported in the highest numbered binding of the layout, if that binding is variable-sized. If the highest numbered binding of the layout has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT then maxVariableDescriptorCount indicates the maximum byte size supported for the binding, if that binding is variable-sized.

If the create info includes a variable-sized descriptor, then supported is determined assuming the requested size of the variable-sized descriptor, and maxVariableDescriptorCount is set to the maximum size of that descriptor that can be successfully created (which is greater than or equal to the requested size passed in). If the create info does not include a variable-sized descriptor or if the VkPhysicalDeviceDescriptorIndexingFeatures::descriptorBindingVariableDescriptorCount feature is not enabled, then maxVariableDescriptorCount is set to zero. For the purposes of this command, a variable-sized descriptor binding with a descriptorCount of zero is treated as if the descriptorCount is one, and thus the binding is not ignored and the maximum descriptor count will be returned. If the layout is not supported, then the value written to maxVariableDescriptorCount is undefined.

Valid Usage (Implicit)

• VUID-VkDescriptorSetVariableDescriptorCountLayoutSupport-sType-sType
sType must be
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_LAYOUT_SUPPORT

The following examples show a shader snippet using two descriptor sets, and application code that creates corresponding descriptor set layouts.
GLSL example

// binding to a single sampled image descriptor in set 0
//
layout (set=0, binding=0) uniform texture2D mySampledImage;

// binding to an array of sampled image descriptors in set 0
//
layout (set=0, binding=1) uniform texture2D myArrayOfSampledImages[12];

// binding to a single uniform buffer descriptor in set 1
//
layout (set=1, binding=0) uniform myUniformBuffer
{
    vec4 myElement[32];
};
SPIR-V example

```assembly
...%1 = OpExtInstImport "GLSL.std.450"
...

OpName %9 "mySampledImage"
OpName %14 "myArrayOfSampledImages"
OpName %18 "myUniformBuffer"
OpMemberName %18 0 "myElement"
OpName %20 ""
OpDecorate %9 DescriptorSet 0
OpDecorate %9 Binding 0
OpDecorate %14 DescriptorSet 0
OpDecorate %14 Binding 1
OpDecorate %17 ArrayStride 16
OpMemberDecorate %18 0 Offset 0
OpDecorate %18 Block
OpDecorate %20 DescriptorSet 1
OpDecorate %20 Binding 0

%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeImage %6 2D 0 0 0 1 Unknown
%8 = OpTypePointer UniformConstant %7
%9 = OpVariable %8 UniformConstant
%10 = OpTypeInt 32 0
%11 = OpConstant %10 12
%12 = OpTypeArray %7 %11
%13 = OpTypePointer UniformConstant %12
%14 = OpVariable %13 UniformConstant
%15 = OpTypeVector %6 4
%16 = OpConstant %10 32
%17 = OpTypeArray %15 %16
%18 = OpTypeStruct %17
%19 = OpTypePointer Uniform %18
%20 = OpVariable %19 Uniform
...
```

API example

```cpp
VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] =
{
    // binding to a single image descriptor
    {
        0,    // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,    // descriptorType
        1,    // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,       // stageFlags
```
// Create info for first descriptor set with two descriptor bindings
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, // sType
    NULL, // pNext
    0, // flags
    2, // bindingCount
    &myDescriptorSetLayoutBinding[0] // pBindings
},

// Create info for second descriptor set with one descriptor binding
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, // sType
    NULL, // pNext
    0, // flags
    1, // bindingCount
}
};

VkDescriptorSetLayout myDescriptorSetLayout[2];

// Create first descriptor set layout
myResult = vkCreateDescriptorSetLayout(myDevice,
    &myDescriptorSetLayoutCreateInfo[0],
To destroy a descriptor set layout, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyDescriptorSetLayout(
    VkDevice device,
    VkDescriptorSetLayout descriptorSetLayout,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the descriptor set layout.
- `descriptorSetLayout` is the descriptor set layout to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-00284
  If `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-00285
  If no `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, `pAllocator` must be NULL
Valid Usage (Implicit)

- VUID-vkDestroyDescriptorSetLayout-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-parameter
  
  If `descriptorSetLayout` is not `VK_NULL_HANDLE`, `descriptorSetLayout` must be a valid `VkDescriptorSetLayout` handle

- VUID-vkDestroyDescriptorSetLayout-pAllocator-parameter
  
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkDestroyDescriptorSetLayout-descriptorSetLayout-parent
  
  If `descriptorSetLayout` is a valid handle, it must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `descriptorSetLayout` must be externally synchronized

14.2.2. Pipeline Layouts

Access to descriptor sets from a pipeline is accomplished through a *pipeline layout*. Zero or more descriptor set layouts and zero or more push constant ranges are combined to form a pipeline layout object describing the complete set of resources that can be accessed by a pipeline. The pipeline layout represents a sequence of descriptor sets with each having a specific layout. This sequence of layouts is used to determine the interface between shader stages and shader resources. Each pipeline is created using a pipeline layout.

Pipeline layout objects are represented by `VkPipelineLayout` handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineLayout)
```

To create a pipeline layout, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreatePipelineLayout(
    VkDevice                                    device,
    const VkPipelineLayoutCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkPipelineLayout*                           pPipelineLayout);
```

- `device` is the logical device that creates the pipeline layout.
- `pCreateInfo` is a pointer to a `VkPipelineLayoutCreateInfo` structure specifying the state of the
pipeline layout object.

- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pPipelineLayout** is a pointer to a `VkPipelineLayout` handle in which the resulting pipeline layout object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreatePipelineLayout-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkCreatePipelineLayout-pCreateInfo-parameter**
  - `pCreateInfo` must be a valid pointer to a valid `VkPipelineLayoutCreateInfo` structure
- **VUID-vkCreatePipelineLayout-pAllocator-parameter**
  - If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreatePipelineLayout-pPipelineLayout-parameter**
  - `pPipelineLayout` must be a valid pointer to a `VkPipelineLayout` handle

### Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The `VkPipelineLayoutCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkPipelineLayoutCreateFlags flags;
    uint32_t                 setLayoutCount;
    const VkDescriptorSetLayout* pSetLayouts;
    uint32_t                 pushConstantRangeCount;
    const VkPushConstantRange* pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
• `setLayoutCount` is the number of descriptor sets included in the pipeline layout.

• `pSetLayouts` is a pointer to an array of `VkDescriptorSetLayout` objects.

• `pushConstantRangeCount` is the number of push constant ranges included in the pipeline layout.

• `pPushConstantRanges` is a pointer to an array of `VkPushConstantRange` structures defining a set of push constant ranges for use in a single pipeline layout. In addition to descriptor set layouts, a pipeline layout also describes how many push constants can be accessed by each stage of the pipeline.

---

**Note**

Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.
Valid Usage

- VUID-VkPipelineLayoutCreateInfo-setLayoutCount-00286
  
  `setLayoutCount` must be less than or equal to `VkPhysicalDeviceLimits::maxBoundDescriptorSets`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03016
  
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSamplers`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03017
  
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` and `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorUniformBuffers`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03018
  
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageBuffers`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03019
  
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSampledImages`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03020
  
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageImages`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03021
  
  The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible to any given shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorInputAttachments`

- VUID-VkPipelineLayoutCreateInfo-descriptorType-02214
  
  The total number of bindings in descriptor set layouts created without the
VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxPerStageDescriptorInlineUniformBlocks

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03022
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindSamplers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03023
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER and VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindUniformBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03024
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03025
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindSampledImages

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03026
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindStorageImages

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03027
  The total number of descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxPerStageDescriptorUpdateAfterBindInputAttachments

- VUID-VkPipelineLayoutCreateInfo-descriptorType-02215
  The total number of bindings with a descriptorType of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to
 VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03028
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetSamplers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03029
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03030
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03031
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffers

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03032
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffersDynamic

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03033
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetSampledImages

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03034
  The total number of descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceLimits::maxDescriptorSetStorageImages
The total number of descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxDescriptorSetInputAttachments`.

The total number of bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set with a `descriptorType` of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceInlineUniformBlockPropertiesEXT::maxDescriptorSetInlineUniformBlocks`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindSamplers`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindUniformBuffers`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindUniformBuffersDynamic`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageBuffers`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageBuffersDynamic`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, and `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindSampledImages`.

The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` and
VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindStorageImages.

- VUID-VkPipelineLayoutCreateInfo-pSetLayouts-03043
  The total number of descriptors of the type VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceDescriptorIndexingProperties::maxDescriptorSetUpdateAfterBindInputAttachments.

- VUID-VkPipelineLayoutCreateInfo-pPushConstantRanges-00292
  Any two elements of pPushConstantRanges must not include the same stage in stageFlags.

- VUID-VkPipelineLayoutCreateInfo-pSetLayouts-00293
  pSetLayouts must not contain more than one descriptor set layout that was created with VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR set.

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03571
  The total number of bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR accessible to any given shader stage across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxPerStageDescriptorAccelerationStructures.

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03572
  The total number of bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxDescriptorSetAccelerationStructures.

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03573
  The total number of bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxDescriptorSetUpdateAfterBindAccelerationStructures.

- VUID-VkPipelineLayoutCreateInfo-descriptorType-03574
  The total number of bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set with a descriptorType of VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR accessible across all shader stages and across all elements of pSetLayouts must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxDescriptorSetUpdateAfterBindAccelerationStructures.
The total number of bindings with a `descriptorType` of `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV` accessible across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxDescriptorSetAccelerationStructures`.

The total number of `pImmutableSamplers` created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` or `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT` across all shader stages and across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceFragmentDensityMap2PropertiesEXT::maxDescriptorSetSubsampledSamplers`.

Any element of `pSetLayouts` must not have been created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE` bit set.

---

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`

If `setLayoutCount` is not `0`, `pSetLayouts` must be a valid pointer to an array of `setLayoutCount` valid `VkDescriptorSetLayout` handles.

If `pushConstantRangeCount` is not `0`, `pPushConstantRanges` must be a valid pointer to an array of `pushConstantRangeCount` valid `VkPushConstantRange` structures.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineLayoutCreateFlags;
```

`VkPipelineLayoutCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkPushConstantRange` structure is defined as:
// Provided by VK_VERSION_1_0

typedef struct VkPushConstantRange {
    VkShaderStageFlags    stageFlags;
    uint32_t              offset;
    uint32_t              size;
} VkPushConstantRange;

- **stageFlags** is a set of stage flags describing the shader stages that will access a range of push constants. If a particular stage is not included in the range, then accessing members of that range of push constants from the corresponding shader stage will return undefined values.

- **offset** and **size** are the start offset and size, respectively, consumed by the range. Both **offset** and **size** are in units of bytes and **must** be a multiple of 4. The layout of the push constant variables is specified in the shader.

**Valid Usage**

- VUID-VkPushConstantRange-offset-00294
  **offset** **must** be less than `VkPhysicalDeviceLimits::maxPushConstantsSize`

- VUID-VkPushConstantRange-offset-00295
  **offset** **must** be a multiple of 4

- VUID-VkPushConstantRange-size-00296
  **size** **must** be greater than 0

- VUID-VkPushConstantRange-size-00297
  **size** **must** be a multiple of 4

- VUID-VkPushConstantRange-size-00298
  **size** **must** be less than or equal to `VkPhysicalDeviceLimits::maxPushConstantsSize` minus **offset**

**Valid Usage (Implicit)**

- VUID-VkPushConstantRange-stageFlags-parameter
  **stageFlags** **must** be a valid combination of `VkShaderStageFlagBits` values

- VUID-VkPushConstantRange-stageFlags-requiredbitmask
  **stageFlags** **must** not be 0

Once created, pipeline layouts are used as part of pipeline creation (see **Pipelines**), as part of binding descriptor sets (see **Descriptor Set Binding**), and as part of setting push constants (see **Push Constant Updates**). Pipeline creation accepts a pipeline layout as input, and the layout **may** be used to map (set, binding, arrayElement) tuples to implementation resources or memory locations within a descriptor set. The assignment of implementation resources depends only on the bindings defined in the descriptor sets that comprise the pipeline layout, and not on any shader source.

All resource variables **statically used** in all shaders in a pipeline **must** be declared with a
(set, binding, arrayElement) that exists in the corresponding descriptor set layout and is of an appropriate descriptor type and includes the set of shader stages it is used by in `stageFlags`. The pipeline layout can include entries that are not used by a particular pipeline, or that are dead-code eliminated from any of the shaders. The pipeline layout allows the application to provide a consistent set of bindings across multiple pipeline compiles, which enables those pipelines to be compiled in a way that the implementation may cheaply switch pipelines without reprogramming the bindings.

Similarly, the push constant block declared in each shader (if present) must only place variables at offsets that are each included in a push constant range with `stageFlags` including the bit corresponding to the shader stage that uses it. The pipeline layout can include ranges or portions of ranges that are not used by a particular pipeline, or for which the variables have been dead-code eliminated from any of the shaders.

There is a limit on the total number of resources of each type that can be included in bindings in all descriptor set layouts in a pipeline layout as shown in Pipeline Layout Resource Limits. The “Total Resources Available” column gives the limit on the number of each type of resource that can be included in bindings in all descriptor sets in the pipeline layout. Some resource types count against multiple limits. Additionally, there are limits on the total number of each type of resource that can be used in any pipeline stage as described in Shader Resource Limits.

### Table 18. Pipeline Layout Resource Limits

<table>
<thead>
<tr>
<th>Total Resources Available</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maxDescriptorSetSamplers</code> or <code>maxDescriptorSetUpdateAfterBindSamplers</code></td>
<td>sampler</td>
</tr>
<tr>
<td><code>maxDescriptorSetSampledImages</code> or <code>maxDescriptorSetUpdateAfterBindSampledImages</code></td>
<td>combined image sampler</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageImages</code> or <code>maxDescriptorSetUpdateAfterBindStorageImages</code></td>
<td>sampled image</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageImages</code> or <code>maxDescriptorSetUpdateAfterBindStorageImages</code></td>
<td>combined image sampler</td>
</tr>
<tr>
<td><code>maxDescriptorSetUniformBuffers</code> or <code>maxDescriptorSetUpdateAfterBindUniformBuffers</code></td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td><code>maxDescriptorSetUniformBuffersDynamic</code> or <code>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</code></td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageBuffers</code> or <code>maxDescriptorSetUpdateAfterBindStorageBuffers</code></td>
<td>storage buffer</td>
</tr>
<tr>
<td><code>maxDescriptorSetStorageBuffersDynamic</code> or <code>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</code></td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td><code>maxDescriptorSetInputAttachments</code> or <code>maxDescriptorSetUpdateAfterBindInputAttachments</code></td>
<td>input attachment</td>
</tr>
</tbody>
</table>

Chapter 14. Resource Descriptors | 941
To destroy a pipeline layout, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyPipelineLayout(
    VkDevice device,
    VkPipelineLayout pipelineLayout,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the pipeline layout.
- `pipelineLayout` is the pipeline layout to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

**Valid Usage**

- VUID-vkDestroyPipelineLayout-pipelineLayout-00299
  If `VkAllocationCallbacks` were provided when `pipelineLayout` was created, a compatible set of callbacks must be provided here.

- VUID-vkDestroyPipelineLayout-pipelineLayout-00300
  If no `VkAllocationCallbacks` were provided when `pipelineLayout` was created, `pAllocator` must be NULL.

- VUID-vkDestroyPipelineLayout-pipelineLayout-02004
  `pipelineLayout` must not have been passed to any `vkCmd*` command for any command buffers that are still in the recording state when `vkDestroyPipelineLayout` is called.
Valid Usage (Implicit)

- **VUID-vkDestroyPipelineLayout-device-parameter**
  
  `device` **must** be a valid `VkDevice` handle

- **VUID-vkDestroyPipelineLayout-pipelineLayout-parameter**
  
  If `pipelineLayout` is not `VK_NULL_HANDLE`, `pipelineLayout` **must** be a valid `VkPipelineLayout` handle

- **VUID-vkDestroyPipelineLayout-pAllocator-parameter**
  
  If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkDestroyPipelineLayout-pPipelineLayout-parent**
  
  If `pipelineLayout` is a valid handle, it **must** have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `pipelineLayout` **must** be externally synchronized

Pipeline Layout Compatibility

Two pipeline layouts are defined to be “compatible for push constants” if they were created with identical push constant ranges. Two pipeline layouts are defined to be “compatible for set N” if they were created with *identically defined* descriptor set layouts for sets zero through N, and if they were created with identical push constant ranges.

When binding a descriptor set (see *Descriptor Set Binding*) to set number N, if the previously bound descriptor sets for sets zero through N-1 were all bound using compatible pipeline layouts, then performing this binding does not disturb any of the lower numbered sets. If, additionally, the previously bound descriptor set for set N was bound using a pipeline layout compatible for set N, then the bindings in sets numbered greater than N are also not disturbed.

Similarly, when binding a pipeline, the pipeline **can** correctly access any previously bound descriptor sets which were bound with compatible pipeline layouts, as long as all lower numbered sets were also bound with compatible layouts.

Layout compatibility means that descriptor sets **can** be bound to a command buffer for use by any pipeline created with a compatible pipeline layout, and without having bound a particular pipeline first. It also means that descriptor sets **can** remain valid across a pipeline change, and the same resources will be accessible to the newly bound pipeline.
Implementor’s Note

A consequence of layout compatibility is that when the implementation compiles a pipeline layout and maps pipeline resources to implementation resources, the mechanism for set \( N \) should only be a function of sets \([0..N]\).

**Note**

Place the least frequently changing descriptor sets near the start of the pipeline layout, and place the descriptor sets representing the most frequently changing resources near the end. When pipelines are switched, only the descriptor set bindings that have been invalidated will need to be updated and the remainder of the descriptor set bindings will remain in place.

The maximum number of descriptor sets that can be bound to a pipeline layout is queried from physical device properties (see `maxBoundDescriptorSets` in **Limits**).
API example

```c
const VkDescriptorSetLayout layouts[] = { layout1, layout2 };

const VkPushConstantRange ranges[] = {
    {
        VK_SHADER_STAGE_VERTEX_BIT, // stageFlags
        0, // offset
        4 // size
    },
    {
        VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
        4, // offset
        4 // size
    }
};

const VkPipelineLayoutCreateInfo createInfo = {
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO, // sType
    NULL, // pNext
    0, // flags
    2, // setLayoutCount
    layouts, // pSetLayouts
    2, // pushConstantRangeCount
    ranges // pPushConstantRanges
};

VkPipelineLayout myPipelineLayout;
myResult = vkCreatePipelineLayout(
    myDevice,
    &createInfo,
    NULL,
    &myPipelineLayout);
```

14.2.3. Allocation of Descriptor Sets

A descriptor pool maintains a pool of descriptors, from which descriptor sets are allocated. Descriptor pools are externally synchronized, meaning that the application must not allocate and/or free descriptor sets from the same pool in multiple threads simultaneously.

Descriptor pools are represented by VkDescriptorPool handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorPool)
```

To create a descriptor pool object, call:
// Provided by VK_VERSION_1_0
VkResult vkCreateDescriptorPool(
    VkDevice                                    device,
    const VkDescriptorPoolCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorPool*                           pDescriptorPool);

• device is the logical device that creates the descriptor pool.

• pCreateInfo is a pointer to a VkDescriptorPoolCreateInfo structure specifying the state of the descriptor pool object.

• pAllocator controls host memory allocation as described in the Memory Allocation chapter.

• pDescriptorPool is a pointer to a VkDescriptorPool handle in which the resulting descriptor pool object is returned.

The created descriptor pool is returned in pDescriptorPool.

Valid Usage (Implicit)

• VUID-vkCreateDescriptorPool-device-parameter
device must be a valid VkDevice handle

• VUID-vkCreateDescriptorPool-pCreateInfo-parameter
pCreateInfo must be a valid pointer to a valid VkDescriptorPoolCreateInfo structure

• VUID-vkCreateDescriptorPool-pAllocator-parameter
If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• VUID-vkCreateDescriptorPool-pDescriptorPool-parameter
pDescriptorPool must be a valid pointer to a VkDescriptorPool handle

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR.Fragmentation_EXT

Additional information about the pool is passed in a VkDescriptorPoolCreateInfo structure:
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorPoolCreateFlags flags;
    uint32_t maxSets;
    uint32_t poolSizeCount;
    const VkDescriptorPoolSize* pPoolSizes;
} VkDescriptorPoolCreateInfo;

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is a bitmask of `VkDescriptorPoolCreateFlagBits` specifying certain supported operations on the pool.
• `maxSets` is the maximum number of descriptor sets that can be allocated from the pool.
• `poolSizeCount` is the number of elements in `pPoolSizes`.
• `pPoolSizes` is a pointer to an array of `VkDescriptorPoolSize` structures, each containing a descriptor type and number of descriptors of that type to be allocated in the pool.

If multiple `VkDescriptorPoolSize` structures appear in the `pPoolSizes` array then the pool will be created with enough storage for the total number of descriptors of each type.

Fragmentation of a descriptor pool is possible and may lead to descriptor set allocation failures. A failure due to fragmentation is defined as failing a descriptor set allocation despite the sum of all outstanding descriptor set allocations from the pool plus the requested allocation requiring no more than the total number of descriptors requested at pool creation. Implementations provide certain guarantees of when fragmentation must not cause allocation failure, as described below.

If a descriptor pool has not had any descriptor sets freed since it was created or most recently reset then fragmentation must not cause an allocation failure (note that this is always the case for a pool created without the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` bit set). Additionally, if all sets allocated from the pool since it was created or most recently reset use the same number of descriptors (of each type) and the requested allocation also uses that same number of descriptors (of each type), then fragmentation must not cause an allocation failure.

If an allocation failure occurs due to fragmentation, an application can create an additional descriptor pool to perform further descriptor set allocations.

If `flags` has the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` bit set, descriptor pool creation may fail with the error `VK_ERROR_FRAGMENTATION` if the total number of descriptors across all pools (including this one) created with this bit set exceeds `maxUpdateAfterBindDescriptorsInAllPools`, or if fragmentation of the underlying hardware resources occurs.

If a `pPoolSizes[i]::type` is `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`, a `VkMutableDescriptorTypeCreateInfoVALVE` struct in the `pNext` chain can be used to specify which mutable descriptor types can be allocated from the pool. If present in the `pNext` chain, `VkMutableDescriptorTypeCreateInfoVALVE::pMutableDescriptorTypeLists[i]` specifies which kind of
VK_DESCRIPTOR_TYPE_MUTABLE_VALVE descriptors can be allocated from this pool entry. If VkMutableDescriptorTypeCreateInfoVALVE does not exist in the pNext chain, or VkMutableDescriptorTypeCreateInfoVALVE::pMutableDescriptorTypeLists[i] is out of range, the descriptor pool allocates enough memory to be able to allocate a VK_DESCRIPTOR_TYPE_MUTABLE_VALVE descriptor with any supported VkDescriptorType as a mutable descriptor. A mutable descriptor can be allocated from a pool entry if the type list in VkDescriptorSetLayoutCreateInfo is a subset of the type list declared in the descriptor pool, or if the pool entry is created without a descriptor type list. Multiple pPoolSizes entries with VK_DESCRIPTOR_TYPE_MUTABLE_VALVE can be declared. When multiple such pool entries are present in pPoolSizes, they specify sets of supported descriptor types which either fully overlap, partially overlap, or are disjoint. Two sets fully overlap if the sets of supported descriptor types are equal. If the sets are not disjoint they partially overlap. A pool entry without a VkMutableDescriptorTypeListVALVE assigned to it is considered to partially overlap any other pool entry which has a VkMutableDescriptorTypeListVALVE assigned to it. The application must ensure that partial overlap does not exist in pPoolSizes.

Note
The requirement of no partial overlap is intended to resolve ambiguity for validation as there is no confusion which pPoolSizes entries will be allocated from. An implementation is not expected to depend on this requirement.

Valid Usage

- VUID-VkDescriptorPoolCreateInfo-maxSets-00301
  maxSets must be greater than 0

- VUID-VkDescriptorPoolCreateInfo-flags-04607
  If flags has the VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE bit set, then the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT bit must not be set

- VUID-VkDescriptorPoolCreateInfo-flags-04608
  If VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE::mutableDescriptorType is not enabled, pPoolSizes must not contain a descriptorType of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE

- VUID-VkDescriptorPoolCreateInfo-flags-04609
  If flags has the VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE bit set, VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE::mutableDescriptorType must be enabled

- VUID-VkDescriptorPoolCreateInfo-pPoolSizes-04787
  If pPoolSizes contains a descriptorType of VK_DESCRIPTOR_TYPE_MUTABLE_VALVE, any other VK_DESCRIPTOR_TYPE_MUTABLE_VALVE element in pPoolSizes must not have sets of supported descriptor types which partially overlap
Valid Usage (Implicit)

- **VUID-VkDescriptorPoolCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO`

- **VUID-VkDescriptorPoolCreateInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkDescriptorPoolInlineUniformBlockCreateInfoEXT` or `VkMutableDescriptorTypeCreateInfoVALVE`.

- **VUID-VkDescriptorPoolCreateInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkDescriptorPoolCreateInfo-flags-parameter**
  
  `flags` must be a valid combination of `VkDescriptorPoolCreateFlagBits` values.

- **VUID-VkDescriptorPoolCreateInfo-pPoolSizes-parameter**
  
  `pPoolSizes` must be a valid pointer to an array of `poolSizeCount` valid `VkDescriptorPoolSize` structures.

- **VUID-VkDescriptorPoolCreateInfo-poolSizeCount-arraylength**
  
  `poolSizeCount` must be greater than 0.

In order to be able to allocate descriptor sets having inline uniform block bindings the descriptor pool must be created with specifying the inline uniform block binding capacity of the descriptor pool, in addition to the total inline uniform data capacity in bytes which is specified through a `VkDescriptorPoolSize` structure with a `descriptorType` value of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`. This can be done by adding a `VkDescriptorPoolInlineUniformBlockCreateInfoEXT` structure to the `pNext` chain of `VkDescriptorPoolCreateInfo`.

The `VkDescriptorPoolInlineUniformBlockCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_inline_uniform_block
typedef struct VkDescriptorPoolInlineUniformBlockCreateInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           maxInlineUniformBlockBindings;
} VkDescriptorPoolInlineUniformBlockCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxInlineUniformBlockBindings` is the number of inline uniform block bindings to allocate.
Valid Usage (Implicit)

- VUID-VkDescriptorPoolInlineUniformBlockCreateInfoEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT`

Bits which can be set in `VkDescriptorPoolCreateInfo::flags` to enable operations on a descriptor pool are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorPoolCreateFlagBits {
    VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT = 0x00000001,
    VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT = 0x00000002,
    // Provided by VK_VALVE_mutable_descriptor_type
    VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE = 0x00000004,
    // Provided by VK_EXT_descriptor_indexing
    VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT =
        VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT,
} VkDescriptorPoolCreateFlagBits;
```

- `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` specifies that descriptor sets can return their individual allocations to the pool, i.e. all of `vkAllocateDescriptorSets`, `vkFreeDescriptorSets`, and `vkResetDescriptorPool` are allowed. Otherwise, descriptor sets allocated from the pool must not be individually freed back to the pool, i.e. only `vkAllocateDescriptorSets` and `vkResetDescriptorPool` are allowed.

- `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` specifies that descriptor sets allocated from this pool can include bindings with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` bit set. It is valid to allocate descriptor sets that have bindings that do not set the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` bit from a pool that has `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` set.

- `VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE` specifies that this descriptor pool and the descriptor sets allocated from it reside entirely in host memory and cannot be bound. Descriptor sets allocated from this pool are partially exempt from the external synchronization requirement in `vkUpdateDescriptorSetWithTemplateKHR` and `vkUpdateDescriptorSets`. Descriptor sets and their descriptors can be updated concurrently in different threads, though the same descriptor must not be updated concurrently by two threads.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkDescriptorPoolCreateFlags;
```

`VkDescriptorPoolCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorPoolCreateFlagBits`.

The `VkDescriptorPoolSize` structure is defined as:
```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorPoolSize {
    VkDescriptorType    type;
    uint32_t            descriptorCount;
} VkDescriptorPoolSize;
```

- **type** is the type of descriptor.
- **descriptorCount** is the number of descriptors of that type to allocate. If **type** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT** then **descriptorCount** is the number of bytes to allocate for descriptors of this type.

**Note**
When creating a descriptor pool that will contain descriptors for combined image samplers of multi-planar formats, an application needs to account for non-trivial descriptor consumption when choosing the **descriptorCount** value, as indicated by **VkSamplerYcbcrConversionImageFormatProperties::combinedImageSamplerDescriptorCount**.

### Valid Usage

- **VUID-VkDescriptorPoolSize-descriptorCount-00302**
  - **descriptorCount** must be greater than 0
- **VUID-VkDescriptorPoolSize-type-02218**
  - If **type** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT** then **descriptorCount** must be a multiple of 4

### Valid Usage (Implicit)

- **VUID-VkDescriptorPoolSize-type-parameter**
  - **type** must be a valid **VkDescriptorType** value

To destroy a descriptor pool, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyDescriptorPool(
    VkDevice                                    device,
    VkDescriptorPool                            descriptorPool,
    const VkAllocationCallbacks*                pAllocator);
```

- **device** is the logical device that destroys the descriptor pool.
- **descriptorPool** is the descriptor pool to destroy.
- **pAllocator** controls host memory allocation as described in the *Memory Allocation* chapter.
When a pool is destroyed, all descriptor sets allocated from the pool are implicitly freed and become invalid. Descriptor sets allocated from a given pool do not need to be freed before destroying that descriptor pool.

**Valid Usage**

- VUID-vkDestroyDescriptorPool-descriptorPool-00303
  All submitted commands that refer to descriptorPool (via any allocated descriptor sets) must have completed execution

- VUID-vkDestroyDescriptorPool-descriptorPool-00304
  If VkAllocationCallbacks were provided when descriptorPool was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyDescriptorPool-descriptorPool-00305
  If no VkAllocationCallbacks were provided when descriptorPool was created, pAllocator must be NULL

**Valid Usage (Implicit)**

- VUID-vkDestroyDescriptorPool-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyDescriptorPool-descriptorPool-parameter
  If descriptorPool is not VK_NULL_HANDLE, descriptorPool must be a valid VkDescriptorPool handle

- VUID-vkDestroyDescriptorPool-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyDescriptorPool-descriptorPool-parent
  If descriptorPool is a valid handle, it must have been created, allocated, or retrieved from device

**Host Synchronization**

- Host access to descriptorPool must be externally synchronized

Descriptor sets are allocated from descriptor pool objects, and are represented by VkDescriptorSet handles:

```cpp
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSet)
```

To allocate descriptor sets from a descriptor pool, call:
VkResult vkAllocateDescriptorSets(VkDevice device, const VkDescriptorSetAllocateInfo* pAllocateInfo, VkDescriptorSet* pDescriptorSets);

- **device** is the logical device that owns the descriptor pool.
- **pAllocateInfo** is a pointer to a VkDescriptorSetAllocateInfo structure describing parameters of the allocation.
- **pDescriptorSets** is a pointer to an array of VkDescriptorSet handles in which the resulting descriptor set objects are returned.

The allocated descriptor sets are returned in **pDescriptorSets**.

When a descriptor set is allocated, the initial state is largely uninitialized and all descriptors are undefined. Descriptors also become undefined if the underlying resource is destroyed. Descriptor sets containing undefined descriptors can still be bound and used, subject to the following conditions:

- For descriptor set bindings created with the VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT bit set, all descriptors in that binding that are dynamically used must have been populated before the descriptor set is consumed.
- For descriptor set bindings created without the VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT bit set, all descriptors in that binding that are statically used must have been populated before the descriptor set is consumed.
- Descriptor bindings with descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT can be undefined when the descriptor set is consumed; though values in that block will be undefined.
- Entries that are not used by a pipeline can have undefined descriptors.

If a call to vkAllocateDescriptorSets would cause the total number of descriptor sets allocated from the pool to exceed the value of VkDescriptorPoolCreateInfo::maxSets used to create pAllocateInfo->descriptorPool, then the allocation may fail due to lack of space in the descriptor pool. Similarly, the allocation may fail due to lack of space if the call to vkAllocateDescriptorSets would cause the number of any given descriptor type to exceed the sum of all the descriptorCount members of each element of VkDescriptorPoolCreateInfo::pPoolSizes with a type equal to that type.

Additionally, the allocation may also fail if a call to vkAllocateDescriptorSets would cause the total number of inline uniform block bindings allocated from the pool to exceed the value of VkDescriptorPoolInlineUniformBlockCreateInfoEXT::maxInlineUniformBlockBindings used to create the descriptor pool.

If the allocation fails due to no more space in the descriptor pool, and not because of system or device memory exhaustion, then VK_ERROR_OUT_OF_POOL_MEMORY must be returned.

vkAllocateDescriptorSets can be used to create multiple descriptor sets. If the creation of any of those descriptor sets fails, then the implementation must destroy all successfully created descriptor set objects from this command, set all entries of the pDescriptorSets array to VK_NULL_HANDLE.
Valid Usage (Implicit)

- VUID-vkAllocateDescriptorSets-device-parameter
  *device* must be a valid *VkDevice* handle

- VUID-vkAllocateDescriptorSets-pAllocateInfo-parameter
  *pAllocateInfo* must be a valid pointer to a valid *VkDescriptorSetAllocateInfo* structure

- VUID-vkAllocateDescriptorSets-pDescriptorSets-parameter
  *pDescriptorSets* must be a valid pointer to an array of *pAllocateInfo-*descriptorSetCount* *VkDescriptorSet* handles

- VUID-vkAllocateDescriptorSets-pAllocateInfo::descriptorSetCount-arraylength
  *pAllocateInfo-*descriptorSetCount* must be greater than 0

Host Synchronization

- Host access to *pAllocateInfo-*descriptorPool* must be externally synchronized

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_FRAGMENTED_POOL
- VK_ERROR_OUT_OF_POOL_MEMORY

The *VkDescriptorSetAllocateInfo* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorSetAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorPool descriptorPool;
    uint32_t descriptorSetCount;
    const VkDescriptorSetLayout* pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

- *sType* is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `descriptorPool` is the pool which the sets will be allocated from.

• `descriptorSetCount` determines the number of descriptor sets to be allocated from the pool.

• `pSetLayouts` is a pointer to an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

### Valid Usage

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-00308  
  Each element of `pSetLayouts` must not have been created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR` set

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-03044  
  If any element of `pSetLayouts` was created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set, `descriptorPool` must have been created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-04610  
  If any element of `pSetLayouts` was created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE` bit set, `descriptorPool` must have been created with the `VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE` flag set

### Valid Usage (Implicit)

- VUID-VkDescriptorSetAllocateInfo-sType-sType  
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO`

- VUID-VkDescriptorSetAllocateInfo-pNext-pNext  
  `pNext` must be `NULL` or a pointer to a valid instance of `VkDescriptorSetVariableDescriptorCountAllocateInfo`

- VUID-VkDescriptorSetAllocateInfo-sType-unique  
  The `sType` value of each struct in the `pNext` chain must be unique

- VUID-VkDescriptorSetAllocateInfo-descriptorPool-parameter  
  `descriptorPool` must be a valid `VkDescriptorPool` handle

- VUID-VkDescriptorSetAllocateInfo-pSetLayouts-parameter  
  `pSetLayouts` must be a valid pointer to an array of `descriptorSetCount` valid `VkDescriptorSetLayout` handles

- VUID-VkDescriptorSetAllocateInfo-descriptorSetCount-arraylength  
  `descriptorSetCount` must be greater than 0

- VUID-VkDescriptorSetAllocateInfo-commonparent  
  Both of `descriptorPool`, and the elements of `pSetLayouts` must have been created, allocated, or retrieved from the same `VkDevice`

If the `pNext` chain of a `VkDescriptorSetAllocateInfo` structure includes a
The `VkDescriptorSetVariableDescriptorCountAllocateInfo` structure is defined as:

```c
typedef struct VkDescriptorSetVariableDescriptorCountAllocateInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t descriptorSetCount;
    const uint32_t* pDescriptorCounts;
} VkDescriptorSetVariableDescriptorCountAllocateInfo;
```

or the equivalent

```c
// Provided by VK_EXT_descriptor_indexing
typedef VkDescriptorSetVariableDescriptorCountAllocateInfo VkDescriptorSetVariableDescriptorCountAllocateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `descriptorSetCount` is zero or the number of elements in `pDescriptorCounts`.
- `pDescriptorCounts` is a pointer to an array of descriptor counts, with each member specifying the number of descriptors in a variable descriptor count binding in the corresponding descriptor set being allocated.

If `descriptorSetCount` is zero or this structure is not included in the `pNext` chain, then the variable lengths are considered to be zero. Otherwise, `pDescriptorCounts[i]` is the number of descriptors in the variable count descriptor binding in the corresponding descriptor set layout. If the variable count descriptor binding in the corresponding descriptor set layout has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `pDescriptorCounts[i]` specifies the binding's capacity in bytes. If `VkDescriptorSetAllocateInfo::pSetLayouts[i]` does not include a variable count descriptor binding, then `pDescriptorCounts[i]` is ignored.

### Valid Usage

- **VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-descriptorSetCount-03045**
  If `descriptorSetCount` is not zero, `descriptorSetCount` must equal `VkDescriptorSetAllocateInfo::descriptorSetCount`

- **VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-pSetLayouts-03046**
  If `VkDescriptorSetAllocateInfo::pSetLayouts[i]` has a variable descriptor count binding, then `pDescriptorCounts[i]` must be less than or equal to the descriptor count specified for that binding when the descriptor set layout was created.
Valid Usage (Implicit)

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO

- VUID-VkDescriptorSetVariableDescriptorCountAllocateInfo-pDescriptorCounts-parameter
  
  If descriptorSetCount is not 0, pDescriptorCounts must be a valid pointer to an array of descriptorSetCount uint32_t values.

To free allocated descriptor sets, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkFreeDescriptorSets(
    VkDevice device,  
    VkDescriptorPool descriptorPool,  
    uint32_t descriptorSetCount,  
    const VkDescriptorSet* pDescriptorSets);
```

- device is the logical device that owns the descriptor pool.
- descriptorPool is the descriptor pool from which the descriptor sets were allocated.
- descriptorSetCount is the number of elements in the pDescriptorSets array.
- pDescriptorSets is a pointer to an array of handles to VkDescriptorSet objects.

After calling vkFreeDescriptorSets, all descriptor sets in pDescriptorSets are invalid.

Valid Usage

- VUID-vkFreeDescriptorSets-pDescriptorSets-00309
  
  All submitted commands that refer to any element of pDescriptorSets must have completed execution.

- VUID-vkFreeDescriptorSets-pDescriptorSets-00310
  
  pDescriptorSets must be a valid pointer to an array of descriptorSetCount VkDescriptorSet handles, each element of which must either be a valid handle or VK_NULL_HANDLE.

- VUID-vkFreeDescriptorSets-descriptorPool-00312
  
  descriptorPool must have been created with the VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT flag.
Valid Usage (Implicit)

- **VUID-vkFreeDescriptorSets-device-parameter**
  
  *device must be a valid* `VkDevice` *handle*

- **VUID-vkFreeDescriptorSets-descriptorPool-parameter**
  
  *descriptorPool must be a valid* `VkDescriptorPool` *handle*

- **VUID-vkFreeDescriptorSets-descriptorSetCount-arraylength**
  
  *descriptorSetCount must be greater than 0*

- **VUID-vkFreeDescriptorSets-descriptorPool-parent**
  
  *descriptorPool must have been created, allocated, or retrieved from* `device`

- **VUID-vkFreeDescriptorSets-pDescriptorSets-parent**
  
  *Each element of pDescriptorSets that is a valid handle must have been created, allocated, or retrieved from* `descriptorPool`

Host Synchronization

- Host access to *descriptorPool must be externally synchronized*

- Host access to each member of *pDescriptorSets must be externally synchronized*

Return Codes

**Success**

- **VK_SUCCESS**

To return all descriptor sets allocated from a given pool to the pool, rather than freeing individual descriptor sets, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkResetDescriptorPool(
    VkDevice device,
    VkDescriptorPool descriptorPool,
    VkDescriptorPoolResetFlags flags);
```

- **device** is the logical device that owns the descriptor pool.

- **descriptorPool** is the descriptor pool to be reset.

- **flags** is reserved for future use.

Resetting a descriptor pool recycles all of the resources from all of the descriptor sets allocated from the descriptor pool back to the descriptor pool, and the descriptor sets are implicitly freed.
Valid Usage

- VUID-vkResetDescriptorPool-descriptorPool-00313
  All uses of descriptorPool (via any allocated descriptor sets) must have completed execution

Valid Usage (Implicit)

- VUID-vkResetDescriptorPool-device-parameter
device must be a valid VkDevice handle

- VUID-vkResetDescriptorPool-descriptorPool-parameter
descriptorPool must be a valid VkDescriptorPool handle

- VUID-vkResetDescriptorPool-flags-zerobitmask
  flags must be 0

- VUID-vkResetDescriptorPool-descriptorPool-parent
descriptorPool must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to descriptorPool must be externally synchronized

- Host access to any VkDescriptorSet objects allocated from descriptorPool must be externally synchronized

Return Codes

Success
- VK_SUCCESS

// Provided by VK_VERSION_1_0
typedef VkFlags VkDescriptorPoolResetFlags;

VkDescriptorPoolResetFlags is a bitmask type for setting a mask, but is currently reserved for future use.

14.2.4. Descriptor Set Updates

Once allocated, descriptor sets can be updated with a combination of write and copy operations. To update descriptor sets, call:
void vkUpdateDescriptorSets(
    VkDevice device,
    uint32_t descriptorWriteCount,
    const VkWriteDescriptorSet* pDescriptorWrites,
    uint32_t descriptorCopyCount,
    const VkCopyDescriptorSet* pDescriptorCopies);

• `device` is the logical device that updates the descriptor sets.

• `descriptorWriteCount` is the number of elements in the `pDescriptorWrites` array.

• `pDescriptorWrites` is a pointer to an array of `VkWriteDescriptorSet` structures describing the descriptor sets to write to.

• `descriptorCopyCount` is the number of elements in the `pDescriptorCopies` array.

• `pDescriptorCopies` is a pointer to an array of `VkCopyDescriptorSet` structures describing the descriptor sets to copy between.

The operations described by `pDescriptorWrites` are performed first, followed by the operations described by `pDescriptorCopies`. Within each array, the operations are performed in the order they appear in the array.

Each element in the `pDescriptorWrites` array describes an operation updating the descriptor set using descriptors for resources specified in the structure.

Each element in the `pDescriptorCopies` array is a `VkCopyDescriptorSet` structure describing an operation copying descriptors between sets.

If the `dstSet` member of any element of `pDescriptorWrites` or `pDescriptorCopies` is bound, accessed, or modified by any command that was recorded to a command buffer which is currently in the recording or executable state, and any of the descriptor bindings that are updated were not created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` or `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT` bits set, that command buffer becomes invalid.

**Valid Usage**

- VUID-vkUpdateDescriptorSets-None-03047
  Descriptor bindings updated by this command which were created without the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` or `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT` bits set must not be used by any command that was recorded to a command buffer which is in the pending state.
Valid Usage (Implicit)

- **VUID-vkUpdateDescriptorSets-device-parameter**
  - `device` must be a valid `VkDevice` handle

- **VUID-vkUpdateDescriptorSets-pDescriptorWrites-parameter**
  - If `descriptorWriteCount` is not 0, `pDescriptorWrites` must be a valid pointer to an array of `descriptorWriteCount` valid `VkWriteDescriptorSet` structures

- **VUID-vkUpdateDescriptorSets-pDescriptorCopies-parameter**
  - If `descriptorCopyCount` is not 0, `pDescriptorCopies` must be a valid pointer to an array of `descriptorCopyCount` valid `VkCopyDescriptorSet` structures

Host Synchronization

- Host access to `pDescriptorWrites[]`.dstSet must be externally synchronized
- Host access to `pDescriptorCopies[]`.dstSet must be externally synchronized

The `VkWriteDescriptorSet` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkWriteDescriptorSet {
    VkStructureType          sType;
    const void*              pNext;
    VkDescriptorSet          dstSet;
    uint32_t                  dstBinding;
    uint32_t                  dstArrayElement;
    uint32_t                  descriptorCount;
    VkDescriptorType         descriptorType;
    const VkDescriptorImageInfo*  pImageInfo;
    const VkDescriptorBufferInfo*  pBufferInfo;
    const VkBufferView*       pTexelBufferView;
} VkWriteDescriptorSet;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `dstSet` is the destination descriptor set to update.
- `dstBinding` is the descriptor binding within that set.
- `dstArrayElement` is the starting element in that array. If the descriptor binding identified by `dstSet` and `dstBinding` has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `dstArrayElement` specifies the starting byte offset within the binding.
- `descriptorCount` is the number of descriptors to update. If the descriptor binding identified by `dstSet` and `dstBinding` has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, then `descriptorCount` specifies the number of bytes to update. Otherwise, `descriptorCount` is one
of
•  the number of elements in pImageInfo
•  the number of elements in pBufferInfo
•  the number of elements in pTexelBufferView
•  a value matching the dataSize member of a VkWriteDescriptorSetInlineUniformBlockEXT structure in the pNext chain
•  a value matching the accelerationStructureCount of a VkWriteDescriptorSetAccelerationStructureKHR structure in the pNext chain

• descriptorType is a VkDescriptorType specifying the type of each descriptor in pImageInfo, pBufferInfo, or pTexelBufferView, as described below. If VkDescriptorSetLayoutBinding for dstSet at dstBinding is not equal to VK_DESCRIPTOR_TYPE_MUTABLE_VALUE, descriptorType must be the same type as that specified in VkDescriptorSetLayoutBinding for dstSet at dstBinding. The type of the descriptor also controls which array the descriptors are taken from.

• pImageInfo is a pointer to an array of VkDescriptorImageInfo structures or is ignored, as described below.

• pBufferInfo is a pointer to an array of VkDescriptorBufferInfo structures or is ignored, as described below.

• pTexelBufferView is a pointer to an array of VkBufferView handles as described in the Buffer Views section or is ignored, as described below.

Only one of pImageInfo, pBufferInfo, or pTexelBufferView members is used according to the descriptor type specified in the descriptorType member of the containing VkWriteDescriptorSet structure, or none of them in case descriptorType is VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT, in which case the source data for the descriptor writes is taken from the VkWriteDescriptorSetInlineUniformBlockEXT structure included in the pNext chain of VkWriteDescriptorSet, or if descriptorType is VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR, in which case the source data for the descriptor writes is taken from the VkWriteDescriptorSetAccelerationStructureKHR structure in the pNext chain of VkWriteDescriptorSet, or if descriptorType is VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV, in which case the source data for the descriptor writes is taken from the VkWriteDescriptorSetAccelerationStructureNV structure in the pNext chain of VkWriteDescriptorSet, as specified below.

If the nullDescriptor feature is enabled, the buffer, acceleration structure, imageView, or bufferView can be VK_NULL_HANDLE. Loads from a null descriptor return zero values and stores and atomics to a null descriptor are discarded. A null acceleration structure descriptor results in the miss shader being invoked.

If the destination descriptor is a mutable descriptor, the active descriptor type for the destination descriptor becomes descriptorType.

If the dstBinding has fewer than descriptorCount array elements remaining starting from dstArrayElement, then the remainder will be used to update the subsequent binding - dstBinding+1 starting at array element zero. If a binding has a descriptorCount of zero, it is skipped. This behavior applies recursively, with the update affecting consecutive bindings as needed to update all
descriptorCount descriptors.

Note
The same behavior applies to bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT where descriptorCount specifies the number of bytes to update while dstArrayElement specifies the starting byte offset, thus in this case if the dstBinding has a smaller byte size than the sum of dstArrayElement and descriptorCount, then the remainder will be used to update the subsequent binding - dstBinding+1 starting at offset zero. This falls out as a special case of the above rule.
Valid Usage

- **VUID-VkWriteDescriptorSet-dstBinding-00315**
  
  *dstBinding must* be less than or equal to the maximum value of *binding* of all *VkDescriptorSetLayoutBinding* structures specified when *dstSet*’s descriptor set layout was created.

- **VUID-VkWriteDescriptorSet-dstBinding-00316**
  
  *dstBinding must* be a binding with a non-zero *descriptorCount*.

- **VUID-VkWriteDescriptorSet-descriptorCount-00317**
  
  All consecutive bindings updated via a single *VkWriteDescriptorSet* structure, except those with a *descriptorCount* of zero, *must* have identical *descriptorType* and *stageFlags*.

- **VUID-VkWriteDescriptorSet-descriptorCount-00318**
  
  All consecutive bindings updated via a single *VkWriteDescriptorSet* structure, except those with a *descriptorCount* of zero, *must* all either use immutable samplers or *must* all not use immutable samplers.

- **VUID-VkWriteDescriptorSet-descriptorType-00319**
  
  *descriptorType must* match the type of *dstBinding* within *dstSet*.

- **VUID-VkWriteDescriptorSet-dstSet-00320**
  
  *dstSet must* be a valid *VkDescriptorSet* handle.

- **VUID-VkWriteDescriptorSet-dstArrayElement-00321**
  
  The sum of *dstArrayElement* and *descriptorCount* *must* be less than or equal to the number of array elements in the descriptor set binding specified by *dstBinding*, and all applicable consecutive bindings, as described by consecutive binding updates.

- **VUID-VkWriteDescriptorSet-descriptorType-02219**
  
  If *descriptorType* is *VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT*, *dstArrayElement must* be an integer multiple of 4.

- **VUID-VkWriteDescriptorSet-descriptorType-02220**
  
  If *descriptorType* is *VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT*, *descriptorCount must* be an integer multiple of 4.

- **VUID-VkWriteDescriptorSet-descriptorType-00322**
  
  If *descriptorType* is *VK_DESCRIPTOR_TYPE_SAMPLER*, *VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER*, *VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE*, *VK_DESCRIPTOR_TYPE_STORAGE_IMAGE*, or *VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT*, *pImageInfo must* be a valid pointer to an array of *descriptorCount* valid *VkDescriptorImageInfo* structures.

- **VUID-VkWriteDescriptorSet-descriptorType-02994**
  
  If *descriptorType* is *VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER* or *VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER*, each element of *pTexelBufferView must* be either a valid *VkBufferView* handle or *VK_NULL_HANDLE*.

- **VUID-VkWriteDescriptorSet-descriptorType-02995**
  
  If *descriptorType* is *VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER* or *VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER* and the nullDescriptor feature is not enabled, each element of *pTexelBufferView must* not be *VK_NULL_HANDLE*.
If `descriptorType` is `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`, `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`, or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`, `pBufferInfo` must be a valid pointer to an array of `descriptorCount` valid `VkDescriptorBufferInfo` structures.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `dstSet` was not allocated with a layout that included immutable samplers for `dstBinding` with `descriptorType`, the sampler member of each element of `pImageInfo` must be a valid `VkSampler` object.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` and the `nullDescriptor` feature is not enabled, the `imageView` member of each element of `pImageInfo` must not be `VK_NULL_HANDLE`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, the `pNext` chain must include a `VkWriteDescriptorSetInlineUniformBlockEXT` structure whose `dataSize` member equals `descriptorCount`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR`, the `pNext` chain must include a `VkWriteDescriptorSetAccelerationStructureKHR` structure whose `accelerationStructureCount` member equals `descriptorCount`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV`, the `pNext` chain must include a `VkWriteDescriptorSetAccelerationStructureNV` structure whose `accelerationStructureCount` member equals `descriptorCount`.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`, then the `imageView` member of each `pImageInfo` element must have been created without a `VkSamplerYcbcrConversionInfo` structure in its `pNext` chain.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and if any element of `pImageInfo` has an `imageView` member that was created with a `VkSamplerYcbcrConversionInfo` structure in its `pNext` chain, then `dstSet` must have been allocated with a layout that included immutable samplers for `dstBinding`, and the corresponding immutable sampler must have been created with an *identically defined* `VkSamplerYcbcrConversionInfo` object.

If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, and `dstSet` was allocated
with a layout that included immutable samplers for dstBinding, then the imageView member of each element of pImageInfo which corresponds to an immutable sampler that enables sampler YC_{Cr} conversion must have been created with a VkSamplerYcbcrConversionInfo structure in its pNext chain with an identically defined VkSamplerYcbcrConversionInfo to the corresponding immutable sampler.

- VUID-VkWriteDescriptorSet-descriptorType-00327
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, pBufferInfo must be a multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment

- VUID-VkWriteDescriptorSet-descriptorType-00328
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, pBufferInfo must be a multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment

- VUID-VkWriteDescriptorSet-descriptorType-00329
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, and the buffer member of any element of pBufferInfo is the handle of a non-sparse buffer, then that buffer must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-VkWriteDescriptorSet-descriptorType-00330
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the range member of each element of pBufferInfo, or the effective range if range is VK_WHOLE_SIZE, must be less than or equal to VkPhysicalDeviceLimits::maxUniformBufferRange

- VUID-VkWriteDescriptorSet-descriptorType-00331
  If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the range member of each element of pBufferInfo, or the effective range if range is VK_WHOLE_SIZE, must be less than or equal to VkPhysicalDeviceLimits::maxStorageBufferRange

- VUID-VkWriteDescriptorSet-descriptorType-00332
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, the VkBuffer that each element of pTexelBufferView was created from must have been created with VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT set

- VUID-VkWriteDescriptorSet-descriptorType-00333
  If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, the VkBuffer that each element of pTexelBufferView was created from must have been created with VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT set
If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`, the `VkBuffer` that each element of `pTexelBufferView` was created from must have been created with `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` set.

- VUID-VkWriteDescriptorSet-descriptorType-00336
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must have been created with the identity swizzle.

- VUID-VkWriteDescriptorSet-descriptorType-00337
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` or `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_STORAGE_BIT` set.

- VUID-VkWriteDescriptorSet-descriptorType-04149
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_SAMPLED_BIT` set.

- VUID-VkWriteDescriptorSet-descriptorType-04150
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` the `imageLayout` member of each element of `pImageInfo` must be a member of the list given in `Sampled Image`.

- VUID-VkWriteDescriptorSet-descriptorType-04151
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` the `imageLayout` member of each element of `pImageInfo` must be a member of the list given in `Combined Image Sampler`.

- VUID-VkWriteDescriptorSet-descriptorType-04152
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` the `imageLayout` member of each element of `pImageInfo` must be a member of the list given in `Input Attachment`.

- VUID-VkWriteDescriptorSet-descriptorType-00338
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set.

- VUID-VkWriteDescriptorSet-descriptorType-00339
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_STORAGE_BIT` set.

- VUID-VkWriteDescriptorSet-descriptorCount-03048
  All consecutive bindings updated via a single `VkWriteDescriptorSet` structure, except those with a `descriptorCount` of zero, must have identical `VkDescriptorBindingFlagBits`.

- VUID-VkWriteDescriptorSet-descriptorType-02752
  If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER`, then `dstSet` must not have been allocated with a layout that included immutable samplers for `dstBinding`.

- VUID-VkWriteDescriptorSet-descriptorType-04611
  If the `VkDescriptorSetLayoutBinding` for `dstSet` at `dstBinding` is `VK_DESCRIPTOR_TYPE_MUTABLE_VALUE`, the new active descriptor type `descriptorType` must exist in the corresponding `pMutableDescriptorTypeLists` list for `dstBinding`.

Chapter 14. Resource Descriptors | 967
Valid Usage (Implicit)

- **VUID-VkWriteDescriptorSet-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`

- **VUID-VkWriteDescriptorSet-pNext-pNext**
  
  Each *pNext* member of any structure (including this one) in the *pNext* chain must be either `NULL` or a pointer to a valid instance of `VkWriteDescriptorSetAccelerationStructureKHR`, `VkWriteDescriptorSetAccelerationStructureNV`, or `VkWriteDescriptorSetInlineUniformBlockEXT`

- **VUID-VkWriteDescriptorSet-sType-unique**
  
  The *sType* value of each struct in the *pNext* chain must be unique

- **VUID-VkWriteDescriptorSet-descriptorType-parameter**
  
  *descriptorType* must be a valid `VkDescriptorType` value

- **VUID-VkWriteDescriptorSet-descriptorCount-arraylength**
  
  *descriptorCount* must be greater than 0

- **VUID-VkWriteDescriptorSet-commonparent**
  
  Both of `dstSet`, and the elements of `pTexelBufferView` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

The type of descriptors in a descriptor set is specified by `VkWriteDescriptorSet::descriptorType`, which must be one of the values:

```c
// Provided by VK_VERSION_1_0
typedef enum VkDescriptorType {
    VK_DESCRIPTOR_TYPE_SAMPLER = 0,
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER = 1,
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE = 2,
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE = 3,
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER = 4,
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER = 5,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER = 6,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER = 7,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC = 8,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC = 9,
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT = 10,
    // Provided by VK_EXT_inline_uniform_block
    VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT = 1000138000,
    // Provided by VK_KHR_acceleration_structure
    VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR = 1000150000,
    // Provided by VK_NV_ray_tracing
    VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV = 1000165000,
    // Provided by VK_VALVE_mutable_descriptor_type
    VK_DESCRIPTOR_TYPE_MUTABLE_VALVE = 1000351000
} VkDescriptorType;
```
VK_DESCRIPTOR_TYPE_SAMPLER specifies a sampler descriptor.

VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER specifies a combined image sampler descriptor.

VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE specifies a sampled image descriptor.

VK_DESCRIPTOR_TYPE_STORAGE_IMAGE specifies a storage image descriptor.

VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER specifies a uniform texel buffer descriptor.

VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER specifies a storage texel buffer descriptor.

VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER specifies a uniform buffer descriptor.

VK_DESCRIPTOR_TYPE_STORAGE_BUFFER specifies a storage buffer descriptor.

VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC specifies a dynamic uniform buffer descriptor.

VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC specifies a dynamic storage buffer descriptor.

VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT specifies an input attachment descriptor.

VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT specifies an inline uniform block.

VK_DESCRIPTOR_TYPE_MUTABLE_VALUE specifies a descriptor of mutable type.

When a descriptor set is updated via elements of VkWriteDescriptorSet, members of pImageInfo, pBufferInfo and pTexelBufferView are only accessed by the implementation when they correspond to descriptor type being defined - otherwise they are ignored. The members accessed are as follows for each descriptor type:

- For VK_DESCRIPTOR_TYPE_SAMPLER, only the sampler member of each element of VkWriteDescriptorSet::pImageInfo is accessed.
- For VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, only the imageView and imageLayout members of each element of VkWriteDescriptorSet::pImageInfo are accessed.
- For VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, all members of each element of VkWriteDescriptorSet::pImageInfo are accessed.
- For VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, all members of each element of VkWriteDescriptorSet::pBufferInfo are accessed.
- For VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, each element of VkWriteDescriptorSet::pTexelBufferView is accessed.

When updating descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT, none of the pImageInfo, pBufferInfo, or pTexelBufferView members are accessed, instead the source data of the descriptor update operation is taken from the VkWriteDescriptorSetInlineUniformBlockEXT structure in the pNext chain of VkWriteDescriptorSet.

When updating descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR, none of the pImageInfo, pBufferInfo, or pTexelBufferView members are accessed, instead the source data of the descriptor update operation is taken from the VkWriteDescriptorSetAccelerationStructureKHR structure in the pNext chain of VkWriteDescriptorSet. When updating descriptors with a descriptorType of VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV, none of the pImageInfo, pBufferInfo, or
pTexelBufferView members are accessed, instead the source data of the descriptor update operation is taken from the VkWriteDescriptorSetAccelerationStructureNV structure in the pNext chain of VkWriteDescriptorSet.

The VkDescriptorBufferInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
} VkDescriptorBufferInfo;
```

- **buffer** is VK_NULL_HANDLE or the buffer resource.
- **offset** is the offset in bytes from the start of buffer. Access to buffer memory via this descriptor uses addressing that is relative to this starting offset.
- **range** is the size in bytes that is used for this descriptor update, or VK_WHOLE_SIZE to use the range from offset to the end of the buffer.

**Note**
When setting range to VK_WHOLE_SIZE, the effective range must not be larger than the maximum range for the descriptor type (maxUniformBufferRange or maxStorageBufferRange). This means that VK_WHOLE_SIZE is not typically useful in the common case where uniform buffer descriptors are suballocated from a buffer that is much larger than maxUniformBufferRange.

For VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC descriptor types, offset is the base offset from which the dynamic offset is applied and range is the static size used for all dynamic offsets.

### Valid Usage

- **VUID-VkDescriptorBufferInfo-offset-00340**  
  offset must be less than the size of buffer

- **VUID-VkDescriptorBufferInfo-range-00341**  
  If range is not equal to VK_WHOLE_SIZE, range must be greater than 0

- **VUID-VkDescriptorBufferInfo-range-00342**  
  If range is not equal to VK_WHOLE_SIZE, range must be less than or equal to the size of buffer minus offset

- **VUID-VkDescriptorBufferInfo-buffer-02998**  
  If the nullDescriptor feature is not enabled, buffer must not be VK_NULL_HANDLE

- **VUID-VkDescriptorBufferInfo-buffer-02999**  
  If buffer is VK_NULL_HANDLE, offset must be zero and range must be VK_WHOLE_SIZE
Valid Usage (Implicit)

- VUID-VkDescriptorBufferInfo-buffer-parameter

  If buffer is not VK_NULL_HANDLE, buffer must be a valid VkBuffer handle

The VkDescriptorImageInfo structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkDescriptorImageInfo {
    VkSampler          sampler;
    VkImageView        imageView;
    VkImageLayout      imageLayout;
} VkDescriptorImageInfo;
```

- **sampler** is a sampler handle, and is used in descriptor updates for types VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER if the binding being updated does not use immutable samplers.

- **imageView** is VK_NULL_HANDLE or an image view handle, and is used in descriptor updates for types VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT.

- **imageLayout** is the layout that the image subresources accessible from imageView will be in at the time this descriptor is accessed. imageLayout is used in descriptor updates for types VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT.

Members of VkDescriptorImageInfo that are not used in an update (as described above) are ignored.
Valid Usage

- **VUID-VkDescriptorImageInfo-imageView-00343**
  *imageView* must not be 2D or 2D array image view created from a 3D image

- **VUID-VkDescriptorImageInfo-imageView-01976**
  If *imageView* is created from a depth/stencil image, the *aspectMask* used to create the *imageView* must include either *VK_IMAGE_ASPECT_DEPTH_BIT* or *VK_IMAGE_ASPECT_STENCIL_BIT* but not both

- **VUID-VkDescriptorImageInfo-imageView-00344**
  *imageView* must match the actual *VkImageLayout* of each subresource accessible from *imageView* at the time this descriptor is accessed as defined by the *image layout matching rules*

- **VUID-VkDescriptorImageInfo-sampler-01564**
  If *sampler* is used and the *VkFormat* of the image is a multi-planar format, the image must have been created with *VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT*, and the *aspectMask* of the *imageView* must be *VK_IMAGE_ASPECT_PLANE_0_BIT*, *VK_IMAGE_ASPECT_PLANE_1_BIT* or (for three-plane formats only) *VK_IMAGE_ASPECT_PLANE_2_BIT*

- **VUID-VkDescriptorImageInfo-mutableComparisonSamplers-04450**
  If the *VK_KHR_portability_subset* extension is enabled, and *VkPhysicalDevicePortabilitySubsetFeaturesKHR::mutableComparisonSamplers* is *VK_FALSE*, then *sampler* must have been created with *VkSamplerCreateInfo::compareEnable* set to *VK_FALSE*.

Valid Usage (Implicit)

- **VUID-VkDescriptorImageInfo-commonparent**
  Both of *imageView*, and *sampler* that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same *VkDevice*

If the *descriptorType* member of *VkWriteDescriptorSet* is *VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT* then the data to write to the descriptor set is specified through a *VkWriteDescriptorSetInlineUniformBlockEXT* structure included in the *pNext* chain of *VkWriteDescriptorSet*.

The *VkWriteDescriptorSetInlineUniformBlockEXT* structure is defined as:

```c
// Provided by VK_EXT_inline_uniform_block
typedef struct VkWriteDescriptorSetInlineUniformBlockEXT {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           dataSize;
    const void*        pData;
} VkWriteDescriptorSetInlineUniformBlockEXT;
```
• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **dataSize** is the number of bytes of inline uniform block data pointed to by **pData**.

• **pData** is a pointer to **dataSize** number of bytes of data to write to the inline uniform block.

### Valid Usage

- **VUID-VkWriteDescriptorSetInlineUniformBlockEXT-dataSize-02222**
  
  * **dataSize** must be an integer multiple of 4

### Valid Usage (Implicit)

- **VUID-VkWriteDescriptorSetInlineUniformBlockEXT-sType-sType**
  
  * **sType** must be **VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT**

- **VUID-VkWriteDescriptorSetInlineUniformBlockEXT-pData-parameter**
  
  * **pData** must be a valid pointer to an array of **dataSize** bytes

- **VUID-VkWriteDescriptorSetInlineUniformBlockEXT-dataSize-arraylength**
  
  * **dataSize** must be greater than 0

The **VkWriteDescriptorSetAccelerationStructureKHR** structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkWriteDescriptorSetAccelerationStructureKHR {
    VkStructureType                      sType;
    const void*                          pNext;
    uint32_t                             accelerationStructureCount;
    const VkAccelerationStructureKHR*    pAccelerationStructures;
} VkWriteDescriptorSetAccelerationStructureKHR;
```

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **accelerationStructureCount** is the number of elements in **pAccelerationStructures**.

• **pAccelerationStructures** are the acceleration structures to update.
Valid Usage

- VUID-VkWriteDescriptorSetAccelerationStructureKHR-accelerationStructureCount-02236
  accelerationStructureCount must be equal to descriptorCount in the extended structure

- VUID-VkWriteDescriptorSetAccelerationStructureKHR-pAccelerationStructures-03579
  Each acceleration structure in pAccelerationStructures must have been created with a type of VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR or VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR

- VUID-VkWriteDescriptorSetAccelerationStructureKHR-pAccelerationStructures-03580
  If the nullDescriptor feature is not enabled, each element of pAccelerationStructures must not be VK_NULL_HANDLE

Valid Usage (Implicit)

- VUID-VkWriteDescriptorSetAccelerationStructureKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR

- VUID-VkWriteDescriptorSetAccelerationStructureKHR-pAccelerationStructures-parameter
  pAccelerationStructures must be a valid pointer to an array of accelerationStructureCount valid or VK_NULL_HANDLE VkAccelerationStructureKHR handles

- VUID-VkWriteDescriptorSetAccelerationStructureKHR-accelerationStructureCount-arraylength
  accelerationStructureCount must be greater than 0

The VkWriteDescriptorSetAccelerationStructureNV structure is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkWriteDescriptorSetAccelerationStructureNV {
    VkStructureType                     sType;
    const void*                         pNext;
    uint32_t                            accelerationStructureCount;
    const VkAccelerationStructureNV*    pAccelerationStructures;
} VkWriteDescriptorSetAccelerationStructureNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **accelerationStructureCount** is the number of elements in pAccelerationStructures.
- **pAccelerationStructures** are the acceleration structures to update.
Valid Usage

- **VUID-VkWriteDescriptorSetAccelerationStructureNV-accelerationStructureCount-03747**
  
  *accelerationStructureCount* must be equal to *descriptorCount* in the extended structure.

- **VUID-VkWriteDescriptorSetAccelerationStructureNV-pAccelerationStructures-03748**
  
  Each acceleration structure in *pAccelerationStructures* must have been created with VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR.

- **VUID-VkWriteDescriptorSetAccelerationStructureNV-pAccelerationStructures-03749**
  
  If the *nullDescriptor* feature is not enabled, each member of *pAccelerationStructures* must not be VK_NULL_HANDLE.

Valid Usage (Implicit)

- **VUID-VkWriteDescriptorSetAccelerationStructureNV-sType-sType**
  
  *sType* must be VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_NV.

- **VUID-VkWriteDescriptorSetAccelerationStructureNV-pAccelerationStructures-parameter**
  
  *pAccelerationStructures* must be a valid pointer to an array of *accelerationStructureCount* valid or VK_NULL_HANDLE VkAccelerationStructureNV handles.

- **VUID-VkWriteDescriptorSetAccelerationStructureNV-accelerationStructureCount-arraylength**
  
  *accelerationStructureCount* must be greater than 0.

The *VkCopyDescriptorSet* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkCopyDescriptorSet {
    VkStructureType    sType;
    const void*        pNext;
    VkDescriptorSet    srcSet;
    uint32_t            srcBinding;
    uint32_t            srcArrayElement;
    VkDescriptorSet    dstSet;
    uint32_t            dstBinding;
    uint32_t            dstArrayElement;
    uint32_t            descriptorCount;
} VkCopyDescriptorSet;
```

- **sType** is the type of this structure.

- **pNext** is NULL or a pointer to a structure extending this structure.

- **srcSet, srcBinding, and srcArrayElement** are the source set, binding, and array element, respectively. If the descriptor binding identified by *srcSet* and *srcBinding* has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT then *srcArrayElement* specifies the starting byte offset within the binding to copy from.

- **dstSet, dstBinding, and dstArrayElement** are the destination set, binding, and array element,
respectively. If the descriptor binding identified by dstSet and dstBinding has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT then dstArrayElement specifies the starting byte offset within the binding to copy to.

• descriptorCount is the number of descriptors to copy from the source to destination. If descriptorCount is greater than the number of remaining array elements in the source or destination binding, those affect consecutive bindings in a manner similar to VkWriteDescriptorSet above. If the descriptor binding identified by srcSet and srcBinding has a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT then descriptorCount specifies the number of bytes to copy and the remaining array elements in the source or destination binding refer to the remaining number of bytes in those.

If the VkDescriptorSetLayoutBinding for dstBinding is VK_DESCRIPTOR_TYPE_MUTABLE_VALUE and srcBinding is not VK_DESCRIPTOR_TYPE_MUTABLE_VALUE, the new active descriptor type becomes the descriptor type of srcBinding. If both VkDescriptorSetLayoutBinding for srcBinding and dstBinding are VK_DESCRIPTOR_TYPE_MUTABLE_VALUE, the active descriptor type in each source descriptor is copied into the corresponding destination descriptor. The active descriptor type can be different for each source descriptor.

**Note**

The intention is that copies to and from mutable descriptors is a simple memcpy. Copies between non-mutable and mutable descriptors are expected to require one memcpy per descriptor to handle the difference in size, but this use case with more than one descriptorCount is considered rare.
Valid Usage

- **VUID-VkCopyDescriptorSet-srcBinding-00345**
  
  **srcBinding** **must** be a valid binding within **srcSet**

- **VUID-VkCopyDescriptorSet-srcArrayElement-00346**
  
  The sum of **srcArrayElement** and **descriptorCount** **must** be less than or equal to the number of array elements in the descriptor set binding specified by **srcBinding**, and all applicable consecutive bindings, as described by **consecutive binding updates**

- **VUID-VkCopyDescriptorSet-dstBinding-00347**
  
  **dstBinding** **must** be a valid binding within **dstSet**

- **VUID-VkCopyDescriptorSet-dstArrayElement-00348**
  
  The sum of **dstArrayElement** and **descriptorCount** **must** be less than or equal to the number of array elements in the descriptor set binding specified by **dstBinding**, and all applicable consecutive bindings, as described by **consecutive binding updates**

- **VUID-VkCopyDescriptorSet-dstBinding-02632**
  
  The type of **dstBinding** within **dstSet** **must** be equal to the type of **srcBinding** within **srcSet**

- **VUID-VkCopyDescriptorSet-srcSet-00349**
  
  If **srcSet** is equal to **dstSet**, then the source and destination ranges of descriptors **must** not overlap, where the ranges **may** include array elements from consecutive bindings as described by **consecutive binding updates**

- **VUID-VkCopyDescriptorSet-srcBinding-02223**
  
  If the descriptor type of the descriptor set binding specified by **srcBinding** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT**, **srcArrayElement** **must** be an integer multiple of 4

- **VUID-VkCopyDescriptorSet-dstBinding-02224**
  
  If the descriptor type of the descriptor set binding specified by **dstBinding** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT**, **dstArrayElement** **must** be an integer multiple of 4

- **VUID-VkCopyDescriptorSet-srcBinding-02225**
  
  If the descriptor type of the descriptor set binding specified by either **srcBinding** or **dstBinding** is **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT**, **descriptorCount** **must** be an integer multiple of 4

- **VUID-VkCopyDescriptorSet-srcSet-01918**
  
  If **srcSet**’s layout was created with the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** flag set, then **dstSet**’s layout **must** also have been created with the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** flag set

- **VUID-VkCopyDescriptorSet-srcSet-01919**
  
  If **srcSet**’s layout was created without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** flag set, then **dstSet**’s layout **must** also have been created without the **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** flag set
If the descriptor pool from which `srcSet` was allocated was created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set, then the descriptor pool from which `dstSet` was allocated must also have been created with the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set.

- **VUID-VkCopyDescriptorSet-srcSet-01921**
  If the descriptor pool from which `srcSet` was allocated was created without the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set, then the descriptor pool from which `dstSet` was allocated must also have been created without the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag set.

- **VUID-VkCopyDescriptorSet-dstSet-02753**
  If the descriptor type of the descriptor set binding specified by `dstBinding` is `VK_DESCRIPTOR_TYPE_SAMPLER`, then `dstSet` must not have been allocated with a layout that included immutable samplers for `dstBinding`.

- **VUID-VkCopyDescriptorSet-dstSet-04612**
  If `VkDescriptorSetLayoutBinding` for `dstSet` at `dstBinding` is `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`, the new active descriptor type must exist in the corresponding `pMutableDescriptorTypeLists` list for `dstBinding` if the new active descriptor type is not `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`.

- **VUID-VkCopyDescriptorSet-srcSet-04613**
  If `VkDescriptorSetLayoutBinding` for `srcSet` at `srcBinding` is `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE` and the `VkDescriptorSetLayoutBinding` for `dstSet` at `dstBinding` is not `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`, the active descriptor type for the source descriptor must match the descriptor type of `dstBinding`.

- **VUID-VkCopyDescriptorSet-dstSet-04614**
  If `VkDescriptorSetLayoutBinding` for `dstSet` at `dstBinding` is `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`, and the new active descriptor type is `VK_DESCRIPTOR_TYPE_MUTABLE_VALVE`, the `pMutableDescriptorTypeLists` for `srcBinding` and `dstBinding` must match exactly.

---

**Valid Usage (Implicit)**

- **VUID-VkCopyDescriptorSet-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET`.

- **VUID-VkCopyDescriptorSet-pNext-pNext**
  `pNext` must be `NULL`.

- **VUID-VkCopyDescriptorSet-srcSet-parameter**
  `srcSet` must be a valid `VkDescriptorSet` handle.

- **VUID-VkCopyDescriptorSet-dstSet-parameter**
  `dstSet` must be a valid `VkDescriptorSet` handle.

- **VUID-VkCopyDescriptorSet-commonparent**
  Both of `dstSet`, and `srcSet` must have been created, allocated, or retrieved from the same `VkDevice`. 
14.2.5. Descriptor Update Templates

A descriptor update template specifies a mapping from descriptor update information in host memory to descriptors in a descriptor set. It is designed to avoid passing redundant information to the driver when frequently updating the same set of descriptors in descriptor sets.

Descriptor update template objects are represented by `VkDescriptorUpdateTemplate` handles:

```c
// Provided by VK_VERSION_1_1
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorUpdateTemplate)
```

or the equivalent

```c
// Provided by VK_KHR_descriptor_update_template
typedef VkDescriptorUpdateTemplate VkDescriptorUpdateTemplateKHR;
```

14.2.6. Descriptor Set Updates with Templates

Updating a large `VkDescriptorSet` array can be an expensive operation since an application must specify one `VkWriteDescriptorSet` structure for each descriptor or descriptor array to update, each of which re-specifies the same state when updating the same descriptor in multiple descriptor sets. For cases when an application wishes to update the same set of descriptors in multiple descriptor sets allocated using the same `VkDescriptorSetLayout`, `vkUpdateDescriptorSetWithTemplate` can be used as a replacement for `vkUpdateDescriptorSets`.

`VkDescriptorUpdateTemplate` allows implementations to convert a set of descriptor update operations on a single descriptor set to an internal format that, in conjunction with `vkUpdateDescriptorSetWithTemplate` or `vkCmdPushDescriptorSetWithTemplateKHR`, can be more efficient compared to calling `vkUpdateDescriptorSets` or `vkCmdPushDescriptorSetKHR`. The descriptors themselves are not specified in the `VkDescriptorUpdateTemplate`, rather, offsets into an application provided pointer to host memory are specified, which are combined with a pointer passed to `vkUpdateDescriptorSetWithTemplate` or `vkCmdPushDescriptorSetWithTemplateKHR`. This allows large batches of updates to be executed without having to convert application data structures into a strictly-defined Vulkan data structure.

To create a descriptor update template, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkCreateDescriptorUpdateTemplate(
    VkDevice                                    device,
    const VkDescriptorUpdateTemplateCreateInfo* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorUpdateTemplate*                 pDescriptorUpdateTemplate);
```

or the equivalent command
Provided by VK_KHR_descriptor_update_template

```c
VkResult vkCreateDescriptorUpdateTemplateKHR(
    VkDevice                                    device,
    const VkDescriptorUpdateTemplateCreateInfo* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorUpdateTemplate*                 pDescriptorUpdateTemplate);
```

- `device` is the logical device that creates the descriptor update template.
- `pCreateInfo` is a pointer to a `VkDescriptorUpdateTemplateCreateInfo` structure specifying the set of descriptors to update with a single call to `vkCmdPushDescriptorSetWithTemplateKHR` or `vkUpdateDescriptorSetWithTemplate`.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pDescriptorUpdateTemplate` is a pointer to a `VkDescriptorUpdateTemplate` handle in which the resulting descriptor update template object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateDescriptorUpdateTemplate-device-parameter**  
  `device` must be a valid `VkDevice` handle
- **VUID-vkCreateDescriptorUpdateTemplate-pCreateInfo-parameter**  
  `pCreateInfo` must be a valid pointer to a valid `VkDescriptorUpdateTemplateCreateInfo` structure
- **VUID-vkCreateDescriptorUpdateTemplate-pAllocator-parameter**  
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateDescriptorUpdateTemplate-pDescriptorUpdateTemplate-parameter**  
  `pDescriptorUpdateTemplate` must be a valid pointer to a `VkDescriptorUpdateTemplate` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDescriptorUpdateTemplateCreateInfo` structure is defined as:
// Provided by VK_VERSION_1_1
typedef struct VkDescriptorUpdateTemplateCreateInfo {
    VkStructureType              sType;
    const void*                   pNext;
    VkDescriptorUpdateTemplateCreateFlags flags;
    uint32_t                      descriptorUpdateEntryCount;
    const VkDescriptorUpdateTemplateEntry* pDescriptorUpdateEntries;
    VkDescriptorUpdateTemplateType templateType;
    VkDescriptorSetLayout         descriptorSetLayout;
    VkPipelineBindPoint          pipelineBindPoint;
    VkPipelineLayout             pipelineLayout;
    uint32_t                      set;
} VkDescriptorUpdateTemplateCreateInfo;

or the equivalent

// Provided by VK_KHR_descriptor_update_template
typedef VkDescriptorUpdateTemplateCreateInfo VkDescriptorUpdateTemplateCreateInfoKHR;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is reserved for future use.
• descriptorUpdateEntryCount is the number of elements in the pDescriptorUpdateEntries array.
• pDescriptorUpdateEntries is a pointer to an array of VkDescriptorUpdateTemplateEntry structures describing the descriptors to be updated by the descriptor update template.
• templateType Specifies the type of the descriptor update template. If set to VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET it can only be used to update descriptor sets with a fixed descriptorSetLayout. If set to VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR it can only be used to push descriptor sets using the provided pipelineBindPoint, pipelineLayout, and set number.
• descriptorSetLayout is the descriptor set layout used to build the descriptor update template. All descriptor sets which are going to be updated through the newly created descriptor update template must be created with a layout that matches (is the same as, or defined identically to) this layout. This parameter is ignored if templateType is not VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET.
• pipelineBindPoint is a VkPipelineBindPoint indicating the type of the pipeline that will use the descriptors. This parameter is ignored if templateType is not VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR
• pipelineLayout is a VkPipelineLayout object used to program the bindings. This parameter is ignored if templateType is not VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR
• set is the set number of the descriptor set in the pipeline layout that will be updated. This parameter is ignored if templateType is not VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR
**Valid Usage**

- **VID-VkDescriptorUpdateTemplateCreateInfo-templateType-00350**
  
  If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET`, `descriptorSetLayout` must be a valid `VkDescriptorSetLayout` handle.

- **VID-VkDescriptorUpdateTemplateCreateInfo-templateType-00351**
  
  If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR`, `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value.

- **VID-VkDescriptorUpdateTemplateCreateInfo-templateType-00352**
  
  If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR`, `pipelineLayout` must be a valid `VkPipelineLayout` handle.

- **VID-VkDescriptorUpdateTemplateCreateInfo-templateType-00353**
  
  If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR`, `set` must be the unique set number in the pipeline layout that uses a descriptor set layout that was created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`.

- **VID-VkDescriptorUpdateTemplateCreateInfo-templateType-04615**
  
  If `templateType` is `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET`, `descriptorSetLayout` must not contain a binding with type `VK_DESCRIPTOR_TYPE_MUTABLE_VALUE`.

**Valid Usage (Implicit)**

- **VID-VkDescriptorUpdateTemplateCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO`.

- **VID-VkDescriptorUpdateTemplateCreateInfo-pNext-pNext**
  
  `pNext` must be `NULL`.

- **VID-VkDescriptorUpdateTemplateCreateInfo-flags-zerobitmask**
  
  `flags` must be `0`.

- **VID-VkDescriptorUpdateTemplateCreateInfo-pDescriptorUpdateEntries-parameter**
  
  `pDescriptorUpdateEntries` must be a valid pointer to an array of `VkDescriptorUpdateTemplateEntry` structures.

- **VID-VkDescriptorUpdateTemplateCreateInfo-templateType-parameter**
  
  `templateType` must be a valid `VkDescriptorUpdateTemplateType` value.

- **VID-VkDescriptorUpdateTemplateCreateInfo-descriptorUpdateEntryCount-arraylength**
  
  `descriptorUpdateEntryCount` must be greater than `0`.

- **VID-VkDescriptorUpdateTemplateCreateInfo-commonparent**
  
  Both of `descriptorSetLayout`, and `pipelineLayout` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

---

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkDescriptorUpdateTemplateCreateFlags;
```
or the equivalent

```c
// Provided by VK_KHR_descriptor_update_template
typedef VkDescriptorUpdateTemplateCreateFlags
VkDescriptorUpdateTemplateCreateFlagsKHR;
```

`VkDescriptorUpdateTemplateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The descriptor update template type is determined by the `VkDescriptorUpdateTemplateCreateInfo::templateType` property, which takes the following values:

```c
// Provided by VK_VERSION_1_1
typedef enum VkDescriptorUpdateTemplateType {
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET = 0,
    VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR = 1,
} VkDescriptorUpdateTemplateType;
```

- `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET` specifies that the descriptor update template will be used for descriptor set updates only.
- `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR` specifies that the descriptor update template will be used for push descriptor updates only.

The `VkDescriptorUpdateTemplateEntry` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkDescriptorUpdateTemplateEntry {
    uint32_t            dstBinding;
    uint32_t            dstArrayElement;
    uint32_t            descriptorCount;
    VkDescriptorType    descriptorType;
    size_t              offset;
    size_t              stride;
} VkDescriptorUpdateTemplateEntry;
```
or the equivalent

```c
// Provided by VK_KHR_descriptor_update_template
typedef VkDescriptorUpdateTemplateEntry VkDescriptorUpdateTemplateEntryKHR;
```

- `dstBinding` is the descriptor binding to update when using this descriptor update template.
- `dstArrayElement` is the starting element in the array belonging to `dstBinding`. If the descriptor binding identified by `dstBinding` has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `dstArrayElement` specifies the starting byte offset to update.
- `descriptorCount` is the number of descriptors to update. If `descriptorCount` is greater than the number of remaining array elements in the destination binding, those affect consecutive bindings in a manner similar to `VkWriteDescriptorSet` above. If the descriptor binding identified by `dstBinding` has a descriptor type of `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then `descriptorCount` specifies the number of bytes to update and the remaining array elements in the destination binding refer to the remaining number of bytes in it.
- `descriptorType` is a `VkDescriptorType` specifying the type of the descriptor.
- `offset` is the offset in bytes of the first binding in the raw data structure.
- `stride` is the stride in bytes between two consecutive array elements of the descriptor update informations in the raw data structure. The actual pointer `ptr` for each array element `j` of update entry `i` is computed using the following formula:

```c
const char *ptr = (const char *)pData + pDescriptorUpdateEntries[i].offset + j * pDescriptorUpdateEntries[i].stride
```

The stride is useful in case the bindings are stored in structs along with other data. If `descriptorType` is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT` then the value of `stride` is ignored and the stride is assumed to be 1, i.e. the descriptor update information for them is always specified as a contiguous range.
Valid Usage

- **VUID-VkDescriptorUpdateTemplateEntry-dstBinding-00354**

  `dstBinding` must be a valid binding in the descriptor set layout implicitly specified when using a descriptor update template to update descriptors.

- **VUID-VkDescriptorUpdateTemplateEntry-dstArrayElement-00355**

  `dstArrayElement` and `descriptorCount` must be less than or equal to the number of array elements in the descriptor set binding implicitly specified when using a descriptor update template to update descriptors, and all applicable consecutive bindings, as described by consecutive binding updates.

- **VUID-VkDescriptorUpdateTemplateEntry-descriptor-02226**

  If `descriptor` type is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `dstArrayElement` must be an integer multiple of 4.

- **VUID-VkDescriptorUpdateTemplateEntry-descriptor-02227**

  If `descriptor` type is `VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT`, `descriptorCount` must be an integer multiple of 4.

Valid Usage (Implicit)

- **VUID-VkDescriptorUpdateTemplateEntry-descriptorType-parameter**

  `descriptorType` must be a valid `VkDescriptorType` value.

To destroy a descriptor update template, call:

```c
// Provided by VK_VERSION_1_1
void vkDestroyDescriptorUpdateTemplate(
    VkDevice device,
    VkDescriptorUpdateTemplate descriptorUpdateTemplate,
    const VkAllocationCallbacks* pAllocator);
```

or the equivalent command

```c
// Provided by VK_KHR_descriptor_update_template
void vkDestroyDescriptorUpdateTemplateKHR(
    VkDevice device,
    VkDescriptorUpdateTemplate descriptorUpdateTemplate,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that has been used to create the descriptor update template.
- `descriptorUpdateTemplate` is the descriptor update template to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
Valid Usage

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorSetLayout-00356
  If VkAllocationCallbacks were provided when descriptorUpdateTemplate was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorSetLayout-00357
  If no VkAllocationCallbacks were provided when descriptorUpdateTemplate was created, pAllocator must be NULL

Valid Usage (Implicit)

- VUID-vkDestroyDescriptorUpdateTemplate-device-parameter
device must be a valid VkDevice handle

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorUpdateTemplate-parameter
  If descriptorUpdateTemplate is not VK_NULL_HANDLE, descriptorUpdateTemplate must be a valid VkDescriptorUpdateTemplate handle

- VUID-vkDestroyDescriptorUpdateTemplate-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyDescriptorUpdateTemplate-descriptorUpdateTemplate-parent
  If descriptorUpdateTemplate is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

- Host access to descriptorUpdateTemplate must be externally synchronized

Once a VkDescriptorUpdateTemplate has been created, descriptor sets can be updated by calling:

```c
// Provided by VK_VERSION_1_1
void vkUpdateDescriptorSetWithTemplate(
    VkDevice device,
    VkDescriptorSet descriptorSet,
    VkDescriptorUpdateTemplate descriptorUpdateTemplate,
    const void* pData);
```
or the equivalent command
// Provided by VK_KHR_descriptor_update_template
void vkUpdateDescriptorSetWithTemplateKHR(
    VkDevice                                    device,                        
    VkDescriptorSet                             descriptorSet,                 
    VkDescriptorUpdateTemplate                  descriptorUpdateTemplate,      
    const void*                                 pData);                         

• device is the logical device that updates the descriptor set.
• descriptorSet is the descriptor set to update
• descriptorUpdateTemplate is a VkDescriptorUpdateTemplate object specifying the update mapping between pData and the descriptor set to update.
• pData is a pointer to memory containing one or more VkDescriptorImageInfo, VkDescriptorBufferInfo, or VkBufferView structures or VkAccelerationStructureKHR or VkAccelerationStructureNV handles used to write the descriptors.

Valid Usage

• VUID-vkUpdateDescriptorSetWithTemplate-pData-01685
  pData must be a valid pointer to a memory containing one or more valid instances of VkDescriptorImageInfo, VkDescriptorBufferInfo, or VkBufferView in a layout defined by descriptorUpdateTemplate when it was created with vkCreateDescriptorUpdateTemplate

Valid Usage (Implicit)

• VUID-vkUpdateDescriptorSetWithTemplate-device-parameter
device must be a valid VkDevice handle
• VUID-vkUpdateDescriptorSetWithTemplate-descriptorSet-parameter
descriptorSet must be a valid VkDescriptorSet handle
• VUID-vkUpdateDescriptorSetWithTemplate-descriptorUpdateTemplate-parameter
descriptorUpdateTemplate must be a valid VkDescriptorUpdateTemplate handle
• VUID-vkUpdateDescriptorSetWithTemplate-descriptorUpdateTemplate-parent
descriptorUpdateTemplate must have been created, allocated, or retrieved from device

Host Synchronization

• Host access to descriptorSet must be externally synchronized

API example

struct AppBufferView {
    VkBufferView bufferView;
    uint32_t     applicationRelatedInformation;
}
struct AppDataStructure {
    VkDescriptorImageInfo  imageInfo;    // a single image info
    VkDescriptorBufferInfo bufferInfoArray[3];  // 3 buffer infos in an array
    AppBufferView          bufferView[2];   // An application defined structure
    // ... some more application related data
};

const VkDescriptorUpdateTemplateEntry descriptorUpdateTemplateEntries[] = {
    // binding to a single image descriptor
    { 0, 0, 1, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, offsetof(AppDataStructure, imageInfo), 0 },
    // binding to an array of buffer descriptors
    { 1, 0, 3, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, offsetof(AppDataStructure, bufferInfoArray), sizeof(VkDescriptorBufferInfo) },
    // binding to an array of buffer views
    { 2, 0, 2, VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, offsetof(AppDataStructure, bufferView) + offsetof(AppBufferView, bufferView), sizeof(AppBufferView) } // stride, bufferViews do not have to be compact
};

// create a descriptor update template for descriptor set updates
const VkDescriptorUpdateTemplateCreateInfo createInfo = {
    988 | Chapter 14. Resource Descriptors
VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO, // sType
NULL, // pNext
0, // flags
descriptorUpdateEntryCount
// descriptorUpdateTemplateEntries,
repDescriptorUpdateEntries,
VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET, // templateType
myLayout, // descriptorSetLayout
0, // pipelineBindPoint,
0, // pipelineLayout,
0, // set, ignored by
given templateType
);

VkDescriptorUpdateTemplate myDescriptorUpdateTemplate;
myResult = vkCreateDescriptorUpdateTemplate(
    myDevice,
    &createInfo,
    NULL,
    &myDescriptorUpdateTemplate);

AppDataStructure appData;

// fill appData here or cache it in your engine
vkUpdateDescriptorSetWithTemplate(myDevice, myDescriptorSet,
    myDescriptorUpdateTemplate, &appData);

14.2.7. Descriptor Set Binding

To bind one or more descriptor sets to a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindDescriptorSets(
    VkCommandBuffer commandBuffer,
    VkPipelineBindPoint pipelineBindPoint,
    VkPipelineLayout layout,
    uint32_t firstSet,
    uint32_t descriptorSetCount,
    const VkDescriptorSet* pDescriptorSets,
    uint32_t dynamicOffsetCount,
    const uint32_t* pDynamicOffsets);
```

- `commandBuffer` is the command buffer that the descriptor sets will be bound to.
- `pipelineBindPoint` is a `VkPipelineBindPoint` indicating the type of the pipeline that will use the descriptors. There is a separate set of bind points for each pipeline type, so binding one does not
disturb the others.

- **layout** is a `VkPipelineLayout` object used to program the bindings.
- **firstSet** is the set number of the first descriptor set to be bound.
- **descriptorSetCount** is the number of elements in the `pDescriptorSets` array.
- **pDescriptorSets** is a pointer to an array of handles to `VkDescriptorSet` objects describing the descriptor sets to bind to.
- **dynamicOffsetCount** is the number of dynamic offsets in the `pDynamicOffsets` array.
- **pDynamicOffsets** is a pointer to an array of `uint32_t` values specifying dynamic offsets.

`vkCmdBindDescriptorSets` causes the sets numbered \([\text{firstSet} .. \text{firstSet}+\text{descriptorSetCount}-1]\) to use the bindings stored in `pDescriptorSets[0..\text{descriptorSetCount}-1]` for subsequent bound pipeline commands set by `pipelineBindPoint`. Any bindings that were previously applied via these sets are no longer valid.

Once bound, a descriptor set affects rendering of subsequent commands that interact with the given pipeline type in the command buffer until either a different set is bound to the same set number, or the set is disturbed as described in Pipeline Layout Compatibility.

A compatible descriptor set **must** be bound for all set numbers that any shaders in a pipeline access, at the time that a draw or dispatch command is recorded to execute using that pipeline. However, if none of the shaders in a pipeline statically use any bindings with a particular set number, then no descriptor set need be bound for that set number, even if the pipeline layout includes a non-trivial descriptor set layout for that set number.

If any of the sets being bound include dynamic uniform or storage buffers, then `pDynamicOffsets` includes one element for each array element in each dynamic descriptor type binding in each set. Values are taken from `pDynamicOffsets` in an order such that all entries for set \(N\) come before set \(N+1\); within a set, entries are ordered by the binding numbers in the descriptor set layouts; and within a binding array, elements are in order. `dynamicOffsetCount` **must** equal the total number of dynamic descriptors in the sets being bound.

The effective offset used for dynamic uniform and storage buffer bindings is the sum of the relative offset taken from `pDynamicOffsets`, and the base address of the buffer plus base offset in the descriptor set. The range of the dynamic uniform and storage buffer bindings is the buffer range as specified in the descriptor set.

Each of the `pDescriptorSets` **must** be compatible with the pipeline layout specified by `layout`. The layout used to program the bindings **must** also be compatible with the pipeline used in subsequent bound pipeline commands with that pipeline type, as defined in the Pipeline Layout Compatibility section.

The descriptor set contents bound by a call to `vkCmdBindDescriptorSets` **may** be consumed at the following times:

- For descriptor bindings created with the `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` bit set, the contents **may** be consumed when the command buffer is submitted to a queue, or during shader execution of the resulting draws and dispatches, or any time in between. Otherwise,
• during host execution of the command, or during shader execution of the resulting draws and
  dispatches, or any time in between.

Thus, the contents of a descriptor set binding must not be altered (overwritten by an update
command, or freed) between the first point in time that it may be consumed, and when the
command completes executing on the queue.

The contents of pDynamicOffsets are consumed immediately during execution of
vkCmdBindDescriptorSets. Once all pending uses have completed, it is legal to update and reuse a
descriptor set.

Valid Usage

• VUID-vkCmdBindDescriptorSets-pDescriptorSets-00358
  Each element of pDescriptorSets must have been allocated with a VkDescriptorSetLayout
  that matches (is the same as, or identically defined as) the VkDescriptorSetLayout at set n
  in layout, where n is the sum of firstSet and the index into pDescriptorSets

• VUID-vkCmdBindDescriptorSets-dynamicOffsetCount-00359
  dynamicOffsetCount must be equal to the total number of dynamic descriptors in
  pDescriptorSets

• VUID-vkCmdBindDescriptorSets-firstSet-00360
  The sum of firstSet and descriptorSetCount must be less than or equal to
  VkPipelineLayoutCreateInfo::setLayoutCount provided when layout was created

• VUID-vkCmdBindDescriptorSets-pipelineBindPoint-00361
  pipelineBindPoint must be supported by the commandBuffer's parent VkCommandPool's queue
  family

• VUID-vkCmdBindDescriptorSets-pDynamicOffsets-01971
  Each element of pDynamicOffsets which corresponds to a descriptor binding with type
  VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must be a multiple of VkPhysicalDeviceLimits
  ::minUniformBufferOffsetAlignment

• VUID-vkCmdBindDescriptorSets-pDynamicOffsets-01972
  Each element of pDynamicOffsets which corresponds to a descriptor binding with type
  VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC must be a multiple of VkPhysicalDeviceLimits
  ::minStorageBufferOffsetAlignment

• VUID-vkCmdBindDescriptorSets-pDescriptorSets-01979
  For each dynamic uniform or storage buffer binding in pDescriptorSets, the sum of the
effective offset, as defined above, and the range of the binding must be less than or equal
to the size of the buffer

• VUID-vkCmdBindDescriptorSets-pDescriptorSets-04616
  Each element of pDescriptorSets must not have been allocated from a VkDescriptorPool
  with the VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE flag set
Valid Usage (Implicit)

- VUID-vkCmdBindDescriptorSets-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdBindDescriptorSets-pipelineBindPoint-parameter
  `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value

- VUID-vkCmdBindDescriptorSets-layout-parameter
  `layout` must be a valid `VkPipelineLayout` handle

- VUID-vkCmdBindDescriptorSets-pDescriptorSets-parameter
  `pDescriptorSets` must be a valid pointer to an array of `descriptorSetCount` valid `VkDescriptorSet` handles

- VUID-vkCmdBindDescriptorSets-pDynamicOffsets-parameter
  If `dynamicOffsetCount` is not 0, `pDynamicOffsets` must be a valid pointer to an array of `dynamicOffsetCount` `uint32_t` values

- VUID-vkCmdBindDescriptorSets-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdBindDescriptorSets-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- VUID-vkCmdBindDescriptorSets-descriptorSetCount-arraylength
  `descriptorSetCount` must be greater than 0

- VUID-vkCmdBindDescriptorSets-commonparent
  Each of `commandBuffer`, `layout`, and the elements of `pDescriptorSets` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

992 | Chapter 14. Resource Descriptors
14.2.8. Push Descriptor Updates

In addition to allocating descriptor sets and binding them to a command buffer, an application can record descriptor updates into the command buffer.

To push descriptor updates into a command buffer, call:

```c
// Provided by VK_KHR_push_descriptor
void vkCmdPushDescriptorSetKHR(
    VkCommandBuffer commandBuffer,
    VkPipelineBindPoint pipelineBindPoint,
    VkPipelineLayout layout,
    uint32_t set,
    uint32_t descriptorWriteCount,
    const VkWriteDescriptorSet* pDescriptorWrites);
```

- `commandBuffer` is the command buffer that the descriptors will be recorded in.
- `pipelineBindPoint` is a `VkPipelineBindPoint` indicating the type of the pipeline that will use the descriptors. There is a separate set of push descriptor bindings for each pipeline type, so binding one does not disturb the others.
- `layout` is a `VkPipelineLayout` object used to program the bindings.
- `set` is the set number of the descriptor set in the pipeline layout that will be updated.
- `descriptorWriteCount` is the number of elements in the `pDescriptorWrites` array.
- `pDescriptorWrites` is a pointer to an array of `VkWriteDescriptorSet` structures describing the descriptors to be updated.

Push descriptors are a small bank of descriptors whose storage is internally managed by the command buffer rather than being written into a descriptor set and later bound to a command buffer. Push descriptors allow for incremental updates of descriptors without managing the lifetime of descriptor sets.

When a command buffer begins recording, all push descriptors are undefined. Push descriptors can be updated incrementally and cause shaders to use the updated descriptors for subsequent bound pipeline commands with the pipeline type set by `pipelineBindPoint` until the descriptor is overwritten, or else until the set is disturbed as described in Pipeline Layout Compatibility. When the set is disturbed or push descriptors with a different descriptor set layout are set, all push descriptors are undefined.

Push descriptors that are statically used by a pipeline must not be undefined at the time that a draw or dispatch command is recorded to execute using that pipeline. This includes immutable sampler descriptors, which must be pushed before they are accessed by a pipeline (the immutable samplers are pushed, rather than the samplers in `pDescriptorWrites`). Push descriptors that are not statically used can remain undefined.

Push descriptors do not use dynamic offsets. Instead, the corresponding non-dynamic descriptor types can be used and the `offset` member of `VkDescriptorBufferInfo` can be changed each time the descriptor is written.
Each element of `pDescriptorWrites` is interpreted as in `VkWriteDescriptorSet`, except the `dstSet` member is ignored.

To push an immutable sampler, use a `VkWriteDescriptorSet` with `dstBinding` and `dstArrayElement` selecting the immutable sampler's binding. If the descriptor type is `VK_DESCRIPTOR_TYPE_SAMPLER`, the `pImageInfo` parameter is ignored and the immutable sampler is taken from the push descriptor set layout in the pipeline layout. If the descriptor type is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `sampler` member of the `pImageInfo` parameter is ignored and the immutable sampler is taken from the push descriptor set layout in the pipeline layout.

### Valid Usage

- **VUID-vkCmdPushDescriptorSetKHR-pipelineBindPoint-00363**  
  `pipelineBindPoint` must be supported by the `commandBuffer`'s parent `VkCommandPool`'s queue family

- **VUID-vkCmdPushDescriptorSetKHR-set-00364**  
  `set` must be less than `VkPipelineLayoutCreateInfo::setLayoutCount` provided when `layout` was created

- **VUID-vkCmdPushDescriptorSetKHR-set-00365**  
  `set` must be the unique set number in the pipeline layout that uses a descriptor set layout that was created with `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`
Valid Usage (Implicit)

- **VUID-vkCmdPushDescriptorSetKHR-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdPushDescriptorSetKHR-pipelineBindPoint-parameter**
  
  `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value.

- **VUID-vkCmdPushDescriptorSetKHR-layout-parameter**
  
  `layout` must be a valid `VkPipelineLayout` handle.

- **VUID-vkCmdPushDescriptorSetKHR-pDescriptorWrites-parameter**
  
  `pDescriptorWrites` must be a valid pointer to an array of `descriptorWriteCount` valid `VkWriteDescriptorSet` structures.

- **VUID-vkCmdPushDescriptorSetKHR-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdPushDescriptorSetKHR-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations.

- **VUID-vkCmdPushDescriptorSetKHR-descriptorWriteCount-arraylength**
  
  `descriptorWriteCount` must be greater than 0.

- **VUID-vkCmdPushDescriptorSetKHR-commonparent**
  
  Both of `commandBuffer`, and `layout` must have been created, allocated, or retrieved from the same `VkDevice`.

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14.2.9. Push Descriptor Updates with Descriptor Update Templates

It is also possible to use a descriptor update template to specify the push descriptors to update. To do so, call:
void vkCmdPushDescriptorSetWithTemplateKHR(
    VkCommandBuffer                             commandBuffer,
    VkDescriptorUpdateTemplate                  descriptorUpdateTemplate,
    VkPipelineLayout                            layout,
    uint32_t                                    set,
    const void*                                 pData);

• **commandBuffer** is the command buffer that the descriptors will be recorded in.

• **descriptorUpdateTemplate** is a descriptor update template defining how to interpret the descriptor information in **pData**.

• **layout** is a **VkPipelineLayout** object used to program the bindings. It **must** be compatible with the layout used to create the **descriptorUpdateTemplate** handle.

• **set** is the set number of the descriptor set in the pipeline layout that will be updated. This **must** be the same number used to create the **descriptorUpdateTemplate** handle.

• **pData** is a pointer to memory containing descriptors for the templated update.

**Valid Usage**

• **VUID-vkCmdPushDescriptorSetWithTemplateKHR-commandBuffer-00366**
  The **pipelineBindPoint** specified during the creation of the descriptor update template **must** be supported by the **commandBuffer**'s parent **VkCommandPool**'s queue family

• **VUID-vkCmdPushDescriptorSetWithTemplateKHR-pData-01686**
  **pData** **must** be a valid pointer to a memory containing one or more valid instances of **VkDescriptorImageInfo**, **VkDescriptorBufferInfo**, or **VkBufferView** in a layout defined by **descriptorUpdateTemplate** when it was created with **vkCreateDescriptorUpdateTemplate**
Valid Usage (Implicit)

- VUID-vkCmdPushDescriptorSetWithTemplateKHR-commandBuffer-parameter
  \texttt{commandBuffer} must be a valid 	exttt{VkCommandBuffer} handle

- VUID-vkCmdPushDescriptorSetWithTemplateKHR-descriptorUpdateTemplate-parameter
  \texttt{descriptorUpdateTemplate} must be a valid 	exttt{VkDescriptorUpdateTemplate} handle

- VUID-vkCmdPushDescriptorSetWithTemplateKHR-layout-parameter
  \texttt{layout} must be a valid 	exttt{VkPipelineLayout} handle

- VUID-vkCmdPushDescriptorSetWithTemplateKHR-commandBuffer-recording
  \texttt{commandBuffer} must be in the \texttt{recording} state

- VUID-vkCmdPushDescriptorSetWithTemplateKHR-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics, or compute operations

- VUID-vkCmdPushDescriptorSetWithTemplateKHR-commonparent
  Each of \texttt{commandBuffer}, \texttt{descriptorUpdateTemplate}, and \texttt{layout} must have been created, allocated, or retrieved from the same \texttt{VkDevice}

Host Synchronization

- Host access to \texttt{commandBuffer} must be externally synchronized

- Host access to the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>
API example

```c
struct AppDataStructure
{
    VkDescriptorImageInfo imageInfo;  // a single image info
    // ... some more application related data
};

const VkDescriptorUpdateTemplateEntry descriptorUpdateTemplateEntries[] =
{
    // binding to a single image descriptor
    { 0, 0, 1, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, offsetof(AppDataStructure, imageInfo), 0 },
};

// create a descriptor update template for push descriptor set updates
const VkDescriptorUpdateTemplateCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO, NULL, 0, VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR, myPipelineLayout, 0,
};

VkDescriptorUpdateTemplate myDescriptorUpdateTemplate;
myResult = vkCreateDescriptorUpdateTemplate(myDevice, &createInfo, NULL, &myDescriptorUpdateTemplate);

AppDataStructure appData;
// fill appData here or cache it in your engine
vkCmdPushDescriptorSetWithTemplateKHR(myCmdBuffer, myDescriptorUpdateTemplate, myPipelineLayout, 0, &appData);
```
14.2.10. Push Constant Updates

As described above in section Pipeline Layouts, the pipeline layout defines shader push constants which are updated via Vulkan commands rather than via writes to memory or copy commands.

*Note*

Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

To update push constants, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdPushConstants(  
    VkCommandBuffer commandBuffer,        // commandBuffer is the command buffer in which the push constant update will be recorded.
    VkPipelineLayout layout,              // layout is the pipeline layout used to program the push constant updates.
    VkShaderStageFlags stageFlags,       // stageFlags is a bitmask of VkShaderStageFlagBits specifying the shader stages that will use the push constants in the updated range.
    uint32_t offset,                     // offset is the start offset of the push constant range to update, in units of bytes.
    uint32_t size,                       // size is the size of the push constant range to update, in units of bytes.
    const void* pValues);               // pValues is a pointer to an array of size bytes containing the new push constant values.
```

When a command buffer begins recording, all push constant values are undefined.

Push constant values can be updated incrementally, causing shader stages in stageFlags to read the new data from pValues for push constants modified by this command, while still reading the previous data for push constants not modified by this command. When a bound pipeline command is issued, the bound pipeline's layout must be compatible with the layouts used to set the values of all push constants in the pipeline layout's push constant ranges, as described in Pipeline Layout Compatibility. Binding a pipeline with a layout that is not compatible with the push constant layout does not disturb the push constant values.

*Note*

As stageFlags needs to include all flags the relevant push constant ranges were created with, any flags that are not supported by the queue family that the VkCommandPool used to allocate commandBuffer was created on are ignored.
Valid Usage

- VUID-vkCmdPushConstants-offset-01795
  For each byte in the range specified by offset and size and for each shader stage in stageFlags, there **must** be a push constant range in layout that includes that byte and that stage

- VUID-vkCmdPushConstants-offset-01796
  For each byte in the range specified by offset and size and for each push constant range that overlaps that byte, stageFlags **must** include all stages in that push constant range's VkPushConstantRange::stageFlags

- VUID-vkCmdPushConstants-offset-00368
  offset **must** be a multiple of 4

- VUID-vkCmdPushConstants-size-00369
  size **must** be a multiple of 4

- VUID-vkCmdPushConstants-offset-00370
  offset **must** be less than VkPhysicalDeviceLimits::maxPushConstantsSize

- VUID-vkCmdPushConstants-size-00371
  size **must** be less than or equal to VkPhysicalDeviceLimits::maxPushConstantsSize minus offset
Valid Usage (Implicit)

- VUID-vkCmdPushConstants-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdPushConstants-layout-parameter
  `layout` must be a valid `VkPipelineLayout` handle

- VUID-vkCmdPushConstants-stageFlags-parameter
  `stageFlags` must be a valid combination of `VkShaderStageFlagBits` values

- VUID-vkCmdPushConstants-stageFlags-requiredbitmask
  `stageFlags` must not be 0

- VUID-vkCmdPushConstants-pValues-parameter
  `pValues` must be a valid pointer to an array of `size` bytes

- VUID-vkCmdPushConstants-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdPushConstants-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- VUID-vkCmdPushConstants-size-arraylength
  `size` must be greater than 0

- VUID-vkCmdPushConstants-commonparent
  Both of `commandBuffer`, and `layout` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

14.3. Physical Storage Buffer Access

To query a 64-bit buffer device address value through which buffer memory can be accessed in a shader, call:
// Provided by VK_KHR_buffer_device_address
VkDeviceAddress vkGetBufferDeviceAddressKHR(
    VkDevice                                    device,
    const VkBufferDeviceAddressInfo*           pInfo);

or the equivalent command

// Provided by VK_EXT_buffer_device_address
VkDeviceAddress vkGetBufferDeviceAddressEXT(
    VkDevice                                    device,
    const VkBufferDeviceAddressInfo*           pInfo);

- **device** is the logical device that the buffer was created on.
- **pInfo** is a pointer to a **VkBufferDeviceAddressInfo** structure specifying the buffer to retrieve an address for.

The 64-bit return value is an address of the start of **pInfo->buffer**. The address range starting at this value and whose size is the size of the buffer can be used in a shader to access the memory bound to that buffer, using the **SPV_KHR_physical_storage_buffer** extension or the equivalent **SPV_EXT_physical_storage_buffer** extension and the **PhysicalStorageBuffer** storage class. For example, this value can be stored in a uniform buffer, and the shader can read the value from the uniform buffer and use it to do a dependent read/write to this buffer. A value of zero is reserved as a “null” pointer and must not be returned as a valid buffer device address. All loads, stores, and atomics in a shader through **PhysicalStorageBuffer** pointers must access addresses in the address range of some buffer.

If the buffer was created with a non-zero value of **VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress** or **VkBufferDeviceAddressCreateInfoEXT::deviceAddress** the return value will be the same address that was returned at capture time.

### Valid Usage

- **VUID-vkGetBufferDeviceAddress-bufferDeviceAddress-03324**
  The **bufferDeviceAddress** or **VkPhysicalDeviceBufferDeviceAddressFeaturesEXT::bufferDeviceAddress** feature must be enabled

- **VUID-vkGetBufferDeviceAddress-device-03325**
  If **device** was created with multiple physical devices, then the **bufferDeviceAddressMultiDevice** or **VkPhysicalDeviceBufferDeviceAddressFeaturesEXT::bufferDeviceAddressMultiDevice** feature must be enabled
Valid Usage (Implicit)

- VUID-vkGetBufferDeviceAddress-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetBufferDeviceAddress-pInfo-parameter
pInfo must be a valid pointer to a valid VkBufferDeviceAddressInfo structure

The VkBufferDeviceAddressInfo structure is defined as:

```c
typedef struct VkBufferDeviceAddressInfo {
    VkStructureType   sType;
    const void*        pNext;
    VkBuffer           buffer;
} VkBufferDeviceAddressInfo;
```

or the equivalent

```c
// Provided by VK_KHR_buffer_device_address
typedef VkBufferDeviceAddressInfo VkBufferDeviceAddressInfoKHR;
```

or the equivalent

```c
// Provided by VK_EXT_buffer_device_address
typedef VkBufferDeviceAddressInfo VkBufferDeviceAddressInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- buffer specifies the buffer whose address is being queried.

Valid Usage

- VUID-VkBufferDeviceAddressInfo-buffer-02600
If buffer is non-sparse and was not created with the VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT flag, then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-VkBufferDeviceAddressInfo-buffer-02601
buffer must have been created with VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT
Valid Usage (Implicit)

- VUID-VkBufferDeviceAddressInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO

- VUID-VkBufferDeviceAddressInfo-pNext-pNext
  pNext must be NULL

- VUID-VkBufferDeviceAddressInfo-buffer-parameter
  buffer must be a valid VkBuffer handle

To query a 64-bit buffer opaque capture address, call:

```
// Provided by VK_KHR_buffer_device_address
uint64_t vkGetBufferOpaqueCaptureAddressKHR(
    VkDevice device,
    const VkBufferDeviceAddressInfo* pInfo);
```

- `device` is the logical device that the buffer was created on.
- `pInfo` is a pointer to a VkBufferDeviceAddressInfo structure specifying the buffer to retrieve an address for.

The 64-bit return value is an opaque capture address of the start of `pInfo->buffer`.

If the buffer was created with a non-zero value of `VkBufferOpaqueCaptureAddressCreateInfo::opaqueCaptureAddress` the return value must be the same address.

Valid Usage

- VUID-vkGetBufferOpaqueCaptureAddress-None-03326
  The bufferDeviceAddress feature must be enabled

- VUID-vkGetBufferOpaqueCaptureAddress-device-03327
  If device was created with multiple physical devices, then the bufferDeviceAddressMultiDevice feature must be enabled

Valid Usage (Implicit)

- VUID-vkGetBufferOpaqueCaptureAddress-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetBufferOpaqueCaptureAddress-pInfo-parameter
  pInfo must be a valid pointer to a valid VkBufferDeviceAddressInfo structure
Chapter 15. Shader Interfaces

When a pipeline is created, the set of shaders specified in the corresponding `Vk*PipelineCreateInfo` structure are implicitly linked at a number of different interfaces.

- **Shader Input and Output Interface**
- **Vertex Input Interface**
- **Fragment Output Interface**
- **Fragment Input Attachment Interface**
- **Ray Tracing Pipeline Interface**
- **Shader Resource Interface**

Interface definitions make use of the following SPIR-V decorations:

- `DescriptorSet` and `Binding`
- `Location`, `Component`, and `Index`
- `Flat`, `NoPerspective`, `Centroid`, and `Sample`
- `Block` and `BufferBlock`
- `InputAttachmentIndex`
- `Offset`, `ArrayStride`, and `MatrixStride`
- `BuiltIn`
- `PassthroughNV`

This specification describes valid uses for Vulkan of these decorations. Any other use of one of these decorations is invalid, with the exception that, when using SPIR-V versions 1.4 and earlier: `Block`, `BufferBlock`, `Offset`, `ArrayStride`, and `MatrixStride` can also decorate types and type members used by variables in the Private and Function storage classes.

15.1. Shader Input and Output Interfaces

When multiple stages are present in a pipeline, the outputs of one stage form an interface with the inputs of the next stage. When such an interface involves a shader, shader outputs are matched against the inputs of the next stage, and shader inputs are matched against the outputs of the previous stage.

All the variables forming the shader input and output interfaces are listed as operands to the `OpEntryPoint` instruction and are declared with the `Input` or `Output` storage classes, respectively, in the SPIR-V module. These generally form the interfaces between consecutive shader stages, regardless of any non-shader stages between the consecutive shader stages.

There are two classes of variables that can be matched between shader stages, built-in variables and user-defined variables. Each class has a different set of matching criteria.

Output variables of a shader stage have undefined values until the shader writes to them or uses the
Initializer operand when declaring the variable.

15.1.1. Built-in Interface Block

Shader built-in variables meeting the following requirements define the built-in interface block. They must

- be explicitly declared (there are no implicit built-ins),
- be identified with a BuiltIn decoration,
- form object types as described in the Built-in Variables section, and
- be declared in a block whose top-level members are the built-ins.

There must be no more than one built-in interface block per shader per interface.

Built-ins must not have any Location or Component decorations.

15.1.2. User-defined Variable Interface

The non-built-in variables listed by OpEntryPoint with the Input or Output storage class form the user-defined variable interface. These must have SPIR-V numerical types or, recursively, composite types of such types. By default, the components of such types have a width of 32 or 64 bits. If an implementation supports storageInputOutput16, components can also have a width of 16 bits. These variables must be identified with a Location decoration and can also be identified with a Component decoration.

15.1.3. Interface Matching

Interface matching rules only apply to built-ins when they are declared as members of the built-in interface block.

Tessellation control and mesh shader per-vertex output variables and blocks, and tessellation control, tessellation evaluation, and geometry shader per-vertex input variables and blocks are required to be declared as arrays, with each element representing input or output values for a single vertex of a multi-vertex primitive. For the purposes of interface matching, the outermost array dimension of such variables and blocks is ignored.

A user-defined output variable is considered to match an input variable in the subsequent stage if the two variables are declared with the same Location and Component decoration and match in type and decoration, except that interpolation decorations are not required to match. XfbBuffer, XfbStride, Offset, and Stream are also not required to match for the purposes of interface matching. For the purposes of interface matching, variables declared without a Component decoration are considered to have a Component decoration of zero.

Note

Matching rules for passthrough geometry shaders are slightly different and are described in the Passthrough Interface Matching section.

Variables or block members declared as structures are considered to match in type if and only if
the structure members match in type, decoration, number, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and size.

At an interface between two non-fragment shader stages, the built-in interface block must match exactly, as described above, except for per-view outputs as described in Mesh Shader Per-View Outputs. At an interface involving the fragment shader inputs, the presence or absence of any built-in output does not affect the interface matching.

At an interface between two shader stages, the user-defined variable interface must match exactly, as described above.

Any input value to a shader stage is well-defined as long as the preceding stage writes to a matching output, as described above.

Additionally, scalar and vector inputs are well-defined if there is a corresponding output satisfying all of the following conditions:

- the input and output match exactly in decoration,
- the output is a vector with the same basic type and has at least as many components as the input, and
- the common component type of the input and output is 16-bit integer or floating-point, or 32-bit integer or floating-point (64-bit component types are excluded).

In this case, the components of the input will be taken from the first components of the output, and any extra components of the output will be ignored.

### 15.1.4. Location Assignment

This section describes location assignments for user-defined variables and how many locations are consumed by a given user-variable type. As mentioned above, some inputs and outputs have an additional level of arrayness relative to other shader inputs and outputs. This outer array level is removed from the type before considering how many locations the type consumes.

The Location value specifies an interface slot comprised of a 32-bit four-component vector conveyed between stages. The Component specifies components within these vector locations. Only types with widths of 16, 32 or 64 are supported in shader interfaces.

Inputs and outputs of the following types consume a single interface location:

- 16-bit scalar and vector types, and
- 32-bit scalar and vector types, and
- 64-bit scalar and 2-component vector types.

64-bit three- and four-component vectors consume two consecutive locations.

If a declared input or output is an array of size $n$ and each element takes $m$ locations, it will be assigned $m \times n$ consecutive locations starting with the location specified.
If the declared input or output is an \( n \times m \) 16-, 32- or 64-bit matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an \( n \)-element array of \( m \)-component vectors.

An \texttt{OpVariable} with a structure type that is not a block must be decorated with a \texttt{Location}.

When an \texttt{OpVariable} with a structure type (either block or non-block) is decorated with a \texttt{Location}, the members in the structure type \textbf{must} not be decorated with a \texttt{Location}. The \texttt{OpVariable}'s members are assigned consecutive locations in declaration order, starting from the first member, which is assigned the location decoration from the \texttt{OpVariable}.

When a block-type \texttt{OpVariable} is declared without a \texttt{Location} decoration, each member in its structure type \textbf{must} be decorated with a \texttt{Location}. Types nested deeper than the top-level members \textbf{must} not have \texttt{Location} decorations.

The locations consumed by block and structure members are determined by applying the rules above in a depth-first traversal of the instantiated members as though the structure or block member were declared as an input or output variable of the same type.

Any two inputs listed as operands on the same \texttt{OpEntryPoint} \textbf{must} not be assigned the same location, either explicitly or implicitly. Any two outputs listed as operands on the same \texttt{OpEntryPoint} \textbf{must} not be assigned the same location, either explicitly or implicitly.

The number of input and output locations available for a shader input or output interface are limited, and dependent on the shader stage as described in Shader Input and Output Locations. All variables in both the \texttt{built-in interface block} and the \texttt{user-defined variable interface} count against these limits. Each effective \texttt{Location} \textbf{must} have a value less than the number of locations available for the given interface, as specified in the "Locations Available" column in Shader Input and Output Locations.

<table>
<thead>
<tr>
<th>Shader Interface</th>
<th>Locations Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertex input</td>
<td>maxVertexInputAttributes</td>
</tr>
<tr>
<td>vertex output</td>
<td>maxVertexOutputComponents / 4</td>
</tr>
<tr>
<td>tessellation control input</td>
<td>maxTessellationControlPerVertexInputComponents / 4</td>
</tr>
<tr>
<td>tessellation control output</td>
<td>maxTessellationControlPerVertexOutputComponents / 4</td>
</tr>
<tr>
<td>tessellation evaluation input</td>
<td>maxTessellationEvaluationInputComponents / 4</td>
</tr>
<tr>
<td>tessellation evaluation output</td>
<td>maxTessellationEvaluationOutputComponents / 4</td>
</tr>
<tr>
<td>geometry input</td>
<td>maxGeometryInputComponents / 4</td>
</tr>
<tr>
<td>geometry output</td>
<td>maxGeometryOutputComponents / 4</td>
</tr>
<tr>
<td>fragment input</td>
<td>maxFragmentInputComponents / 4</td>
</tr>
<tr>
<td>fragment output</td>
<td>maxFragmentOutputAttachments</td>
</tr>
</tbody>
</table>
15.1.5. Component Assignment

The Component decoration allows the Location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable or block member starting at component N will consume components N, N+1, N+2, ... up through its size. For 16-, and 32-bit types, it is invalid if this sequence of components gets larger than 3. A scalar 64-bit type will consume two of these components in sequence, and a two-component 64-bit vector type will consume all four components available within a location. A three- or four-component 64-bit vector type must not specify a Component decoration. A three-component 64-bit vector type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations.

A scalar or two-component 64-bit data type must not specify a Component decoration of 1 or 3. A Component decoration must not be specified for any type that is not a scalar or vector.

15.2. Vertex Input Interface

When the vertex stage is present in a pipeline, the vertex shader input variables form an interface with the vertex input attributes. The vertex shader input variables are matched by the Location and Component decorations to the vertex input attributes specified in the pVertexInputState member of the VkGraphicsPipelineCreateInfo structure.

The vertex shader input variables listed by OpEntryPoint with the Input storage class form the vertex input interface. These variables must be identified with a Location decoration and can also be identified with a Component decoration.

For the purposes of interface matching: variables declared without a Component decoration are considered to have a Component decoration of zero. The number of available vertex input locations is given by the maxVertexInputAttributes member of the VkPhysicalDeviceLimits structure.

See Attribute Location and Component Assignment for details.

All vertex shader inputs declared as above must have a corresponding attribute and binding in the pipeline.

15.3. Fragment Output Interface

When the fragment stage is present in a pipeline, the fragment shader outputs form an interface with the output attachments of the current subpass. The fragment shader output variables are matched by the Location and Component decorations to the color attachments specified in the pColorAttachments array of the VkSubpassDescription structure describing the subpass that the fragment shader is executed in.

The fragment shader output variables listed by OpEntryPoint with the Output storage class form the
**fragment output interface.** These variables must be identified with a Location decoration. They can also be identified with a Component decoration and/or an Index decoration. For the purposes of interface matching: variables declared without a Component decoration are considered to have a Component decoration of zero, and variables declared without an Index decoration are considered to have an Index decoration of zero.

A fragment shader output variable identified with a Location decoration of \(i\) is directed to the color attachment indicated by \(p\text{ColorAttachments}[i]\), after passing through the blending unit as described in Blending, if enabled. Locations are consumed as described in Location Assignment. The number of available fragment output locations is given by the maxFragmentOutputAttachments member of the VkPhysicalDeviceLimits structure.

Components of the output variables are assigned as described in Component Assignment. Output components identified as 0, 1, 2, and 3 will be directed to the R, G, B, and A inputs to the blending unit, respectively, or to the output attachment if blending is disabled. If two variables are placed within the same location, they must have the same underlying type (floating-point or integer). The input values to blending or color attachment writes are undefined for components which do not correspond to a fragment shader output.

Fragment outputs identified with an Index of zero are directed to the first input of the blending unit associated with the corresponding Location. Outputs identified with an Index of one are directed to the second input of the corresponding blending unit.

No component aliasing of output variables is allowed, that is there must not be two output variables which have the same location, component, and index, either explicitly declared or implied.

Output values written by a fragment shader must be declared with either OpTypeFloat or OpTypeInt, and a Width of 32. If storageInputOutput16 is supported, output values written by a fragment shader can be also declared with either OpTypeFloat or OpTypeInt and a Width of 16. Composites of these types are also permitted. If the color attachment has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in Conversion from Floating-Point to Normalized Fixed-Point; If the color attachment has an integer format, color values are assumed to be integers and converted to the bit-depth of the target. Any value that cannot be represented in the attachment’s format is undefined. For any other attachment format no conversion is performed. If the type of the values written by the fragment shader do not match the format of the corresponding color attachment, the resulting values are undefined for those components.

### 15.4. Fragment Input Attachment Interface

When a fragment stage is present in a pipeline, the fragment shader pass inputs form an interface with the input attachments of the current pass. The fragment shader pass input variables are matched by InputAttachmentIndex decorations to the input attachments specified in the pInputAttachments array of the VkSubpassDescription structure describing the subpass that the fragment shader is executed in.

The fragment shader pass input variables with the UniformConstant storage class and a decoration of InputAttachmentIndex that are statically used by OpEntryPoint form the fragment input attachment interface. These variables must be declared with a type of OpTypeImage, a Dim operand of
SubpassData, and a Sampled operand of 2.

A subpass input variable identified with an InputAttachmentIndex decoration of $i$ reads from the input attachment indicated by $pInputAttachments[i]$ member of VkSubpassDescription. If the subpass input variable is declared as an array of size $N$, it consumes $N$ consecutive input attachments, starting with the index specified. There must not be more than one input variable with the same InputAttachmentIndex whether explicitly declared or implied by an array declaration. The number of available input attachment indices is given by the maxPerStageDescriptorInputAttachments member of the VkPhysicalDeviceLimits structure.

Variables identified with the InputAttachmentIndex must only be used by a fragment stage. The basic data type (floating-point, integer, unsigned integer) of the subpass input must match the basic format of the corresponding input attachment, or the values of subpass loads from these variables are undefined.

See Input Attachment for more details.

15.5. Ray Tracing Pipeline Interface

Ray tracing pipelines may have more stages than other pipelines with multiple instances of each stage and more dynamic interactions between the stages, but still have interface structures that obey the same general rules as interfaces between shader stages in other pipelines. The three types of inter-stage interface variables for ray tracing pipelines are:

- Ray payloads which contain data tracked for the entire lifetime of the ray.
- Hit attributes which contain data about a specific hit for the duration of its processing.
- Callable data for passing data into and out of a callable shader.

Ray payloads and callable data are used in explicit shader call instructions, so they have an incoming variant to distinguish the parameter passed to the invocation from any other payloads or data being used by subsequent shader call instructions.

An interface structure used between stages must match between the stages using it. Specifically:

- The hit attribute structure read in an any-hit or closest hit shader must be the same structure as the hit attribute structure written in the corresponding intersection shader in the same hit group.
- The incoming callable data for a callable shader must be the same structure as the callable data referenced by the execute callable instruction in the calling shader.
- The ray payload for a shader invoked by a trace ray command must be the same structure for all shader stages using the payload for that ray.

Any shader with an incoming ray payload, incoming callable data, or hit attribute must only declare one variable of that type.

Table 20. Ray Pipeline Shader Interface
### 15.6. Shader Resource Interface

When a shader stage accesses buffer or image resources, as described in the Resource Descriptors section, the shader resource variables must be matched with the pipeline layout that is provided at pipeline creation time.

The set of shader variables that form the shader resource interface for a stage are the variables statically used by that stage's `OpEntryPoint` with a storage class of `Uniform`, `UniformConstant`, `StorageBuffer`, or `PushConstant`. For the fragment shader, this includes the fragment input attachment interface.

The shader resource interface consists of two sub-interfaces: the push constant interface and the descriptor set interface.

#### 15.6.1. Push Constant Interface

The shader variables defined with a storage class of `PushConstant` that are statically used by the shader entry points for the pipeline define the push constant interface. They must be:

- typed as `OpTypeStruct`,
- identified with a `Block` decoration, and
- laid out explicitly using the `Offset`, `ArrayStride`, and `MatrixStride` decorations as specified in Offset and Stride Assignment.

There must be no more than one push constant block statically used per shader entry point.

Each statically used member of a push constant block must be placed at an `Offset` such that the entire member is entirely contained within the `VkPushConstantRange` for each `OpEntryPoint` that uses it, and the `stageFlags` for that range must specify the appropriate `VkShaderStageFlagBits` for that stage. The `Offset` decoration for any member of a push constant block must not cause the space required for that member to extend outside the range `[0, maxPushConstantsSize)`.

Any member of a push constant block that is declared as an array must only be accessed with dynamically uniform indices.

<table>
<thead>
<tr>
<th>Shader Stage</th>
<th>Ray Payload</th>
<th>Incoming Ray Payload</th>
<th>Hit Attribute</th>
<th>Callable Data</th>
<th>Incoming Callable Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray Generation</td>
<td>r/w</td>
<td>r/w</td>
<td>r/w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intersection</td>
<td></td>
<td></td>
<td></td>
<td>r/w</td>
<td></td>
</tr>
<tr>
<td>Any-Hit</td>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Closest Hit</td>
<td>r/w</td>
<td>r/w</td>
<td>r</td>
<td>r/w</td>
<td></td>
</tr>
<tr>
<td>Miss</td>
<td>r/w</td>
<td>r/w</td>
<td>r/w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Callable</td>
<td></td>
<td></td>
<td>r/w</td>
<td>r/w</td>
<td></td>
</tr>
</tbody>
</table>
15.6.2. Descriptor Set Interface

The descriptor set interface is comprised of the shader variables with the storage class of StorageBuffer, Uniform or UniformConstant (including the variables in the fragment input attachment interface) that are statically used by the shader entry points for the pipeline.

These variables must have DescriptorSet and Binding decorations specified, which are assigned and matched with the VkDescriptorSetLayout objects in the pipeline layout as described in DescriptorSet and Binding Assignment.

The Image Format of an OpTypeImage declaration must not be Unknown, for variables which are used for OpImageRead, OpImageSparseRead, or OpImageWrite operations, except under the following conditions:

- For OpImageWrite, if the shaderStorageImageWriteWithoutFormat feature is enabled and the shader module declares the StorageImageWriteWithoutFormat capability.
- For OpImageRead or OpImageSparseRead, if the shaderStorageImageReadWithoutFormat feature is enabled and the shader module declares the StorageImageReadWithoutFormat capability.
- For OpImageRead, if Dim is SubpassData (indicating a read from an input attachment).

The Image Format of an OpTypeImage declaration must not be Unknown, for variables which are used for OpAtomic* operations.

Variables identified with the Uniform storage class are used to access transparent buffer backed resources. Such variables must be:

- typed as OpTypeStruct, or an array of this type,
- identified with a Block or BufferBlock decoration, and
- laid out explicitly using the Offset, ArrayStride, and MatrixStride decorations as specified in Offset and Stride Assignment.

Variables identified with the StorageBuffer storage class are used to access transparent buffer backed resources. Such variables must be:

- typed as OpTypeStruct, or an array of this type,
- identified with a Block decoration, and
- laid out explicitly using the Offset, ArrayStride, and MatrixStride decorations as specified in Offset and Stride Assignment.

The Offset decoration for any member of a Block-decorated variable in the Uniform storage class must not cause the space required for that variable to extend outside the range [0, maxUniformBufferRange). The Offset decoration for any member of a Block-decorated variable in the StorageBuffer storage class must not cause the space required for that variable to extend outside the range [0, maxStorageBufferRange).

Variables identified with the Uniform storage class can also be used to access transparent descriptor set backed resources when the variable is assigned to a descriptor set layout binding with a descriptorType of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT. In this case the variable must be
typed as `OpTypeStruct` and **cannot** be aggregated into arrays of that type. Further, the Offset decoration for any member of such a variable **must** not cause the space required for that variable to extend outside the range \([0, \text{maxInlineUniformBlockSize}]\).

Variables identified with a storage class of `UniformConstant` and a decoration of `InputAttachmentIndex` **must** be declared as described in Fragment Input Attachment Interface.

SPIR-V variables decorated with a descriptor set and binding that identify a combined image sampler descriptor **can** have a type of `OpTypeImage`, `OpTypeSampler (Sampled=1)`, or `OpTypeSampledImage`.

Arrays of any of these types **can** be indexed with *constant integral expressions*. The following features **must** be enabled and capabilities **must** be declared in order to index such arrays with dynamically uniform or non-uniform indices:

- **Storage images (except storage texel buffers and input attachments):**
  - Dynamically uniform: `shaderStorageImageArrayDynamicIndexing` and `StorageImageArrayDynamicIndexing`
  - Non-uniform: `shaderStorageImageArrayNonUniformIndexing` and `StorageImageArrayNonUniformIndexing`

- **Storage texel buffers:**
  - Dynamically uniform: `shaderStorageTexelBufferArrayDynamicIndexing` and `StorageTexelBufferArrayDynamicIndexing`
  - Non-uniform: `shaderStorageTexelBufferArrayNonUniformIndexing` and `StorageTexelBufferArrayNonUniformIndexing`

- **Input attachments:**
  - Dynamically uniform: `shaderInputAttachmentArrayDynamicIndexing` and `InputAttachmentArrayDynamicIndexing`
  - Non-uniform: `shaderInputAttachmentArrayNonUniformIndexing` and `InputAttachmentArrayNonUniformIndexing`

- **Sampled images (except uniform texel buffers), samplers and combined image samplers:**
  - Dynamically uniform: `shaderSampledImageArrayDynamicIndexing` and `SampledImageArrayDynamicIndexing`
  - Non-uniform: `shaderSampledImageArrayNonUniformIndexing` and `SampledImageArrayNonUniformIndexing`

- **Uniform texel buffers:**
  - Dynamically uniform: `shaderUniformTexelBufferArrayDynamicIndexing` and `UniformTexelBufferArrayDynamicIndexing`
  - Non-uniform: `shaderUniformTexelBufferArrayNonUniformIndexing` and `UniformTexelBufferArrayNonUniformIndexing`

- **Uniform buffers:**
  - Dynamically uniform: `shaderUniformBufferArrayDynamicIndexing` and `UniformBufferArrayDynamicIndexing`
  - Non-uniform: `shaderUniformBufferArrayNonUniformIndexing` and `UniformBufferArrayNonUniformIndexing`

- **Storage buffers:**
Dynamically uniform: shaderStorageBufferArrayDynamicIndexing and StorageBufferArrayDynamicIndexing

Non-uniform: shaderStorageBufferArrayNonUniformIndexing and StorageBufferArrayNonUniformIndexing

Acceleration structures:
- Dynamically uniform: Always supported.
- Non-uniform: Always supported.

If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the corresponding non-uniform indexing feature must be enabled and the capability must be declared. If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is loaded from an array element with a non-constant index, then the corresponding dynamic or non-uniform indexing feature must be enabled and the capability must be declared.

If the combined image sampler enables sampler Y'CbCr conversion or samples a subsampled image, it must be indexed only by constant integral expressions when aggregated into arrays in shader code, irrespective of the shaderSampledImageArrayDynamicIndexing feature.

Table 21. Shader Resource and Descriptor Type Correspondence

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Descriptor Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampler</td>
<td>VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
</tr>
<tr>
<td>sampled image</td>
<td>VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
</tr>
<tr>
<td>storage image</td>
<td>VK_DESCRIPTOR_TYPE_STORAGE_IMAGE</td>
</tr>
<tr>
<td>combined image sampler</td>
<td>VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER</td>
</tr>
<tr>
<td>uniform texel buffer</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER</td>
</tr>
<tr>
<td>storage texel buffer</td>
<td>VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER</td>
</tr>
<tr>
<td>uniform buffer</td>
<td>VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC</td>
</tr>
<tr>
<td>storage buffer</td>
<td>VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC</td>
</tr>
<tr>
<td>input attachment</td>
<td>VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT</td>
</tr>
<tr>
<td>inline uniform block</td>
<td>VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT</td>
</tr>
<tr>
<td>acceleration structure</td>
<td>VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR or VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV</td>
</tr>
</tbody>
</table>

Table 22. Shader Resource and Storage Class Correspondence

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Storage Class</th>
<th>Type</th>
<th>Decoration(s)¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampler</td>
<td>UniformConstant</td>
<td>OpTypeSampler</td>
<td></td>
</tr>
<tr>
<td>sampled image</td>
<td>UniformConstant</td>
<td>OpTypeImage (Sampled=1)</td>
<td></td>
</tr>
</tbody>
</table>

¹Decoration(s)

Chapter 15. Shader Interfaces | 1015
<table>
<thead>
<tr>
<th>Resource type</th>
<th>Storage Class</th>
<th>Type</th>
<th>Decoration(s)¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage image</td>
<td>UniformConstant</td>
<td>OpTypeImage (Sampled=2)</td>
<td></td>
</tr>
<tr>
<td>combined image</td>
<td>UniformConstant</td>
<td>OpTypeSampledImage</td>
<td></td>
</tr>
<tr>
<td>sampler</td>
<td></td>
<td>OpTypeImage (Sampled=1)</td>
<td></td>
</tr>
<tr>
<td>uniform texel buffer</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim=Buffer, Sampled=1)</td>
<td></td>
</tr>
<tr>
<td>storage texel buffer</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim=Buffer, Sampled=2)</td>
<td></td>
</tr>
<tr>
<td>uniform buffer</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>Block, Offset, (MatrixStride)</td>
</tr>
<tr>
<td>storage buffer</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>BufferBlock, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td></td>
<td>StorageBuffer</td>
<td>OpTypeStruct</td>
<td>Block, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td>input attachment</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim =SubpassData, Sampled=2)</td>
<td>InputAttachmentIndex</td>
</tr>
<tr>
<td>inline uniform block</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>Block, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td>acceleration structure</td>
<td>UniformConstant</td>
<td>OpTypeAccelerationStructureKHR</td>
<td></td>
</tr>
</tbody>
</table>

¹ in addition to DescriptorSet and Binding

### 15.6.3. DescriptorSet and Binding Assignment

A variable decorated with a `DescriptorSet` decoration of s and a `Binding` decoration of b indicates that this variable is associated with the `VkDescriptorSetLayoutBinding` that has a `binding` equal to b in `pSetLayouts[s]` that was specified in `VkPipelineLayoutCreateInfo`.

`DescriptorSet` decoration values **must** be between zero and `maxBoundDescriptorSets` minus one, inclusive. `Binding` decoration values **can** be any 32-bit unsigned integer value, as described in `Descriptor Set Layout`. Each descriptor set has its own binding name space.

If the `Binding` decoration is used with an array, the entire array is assigned that binding value. The array **must** be a single-dimensional array and size of the array **must** be no larger than the number of descriptors in the binding. If the array is runtime-sized, then array elements greater than or equal to the size of that binding in the bound descriptor set **must** not be used. If the array is runtime-sized, the `runtimeDescriptorArray` feature **must** be enabled and the `RuntimeDescriptorArray` capability **must** be declared. The index of each element of the array is referred to as the `arrayElement`. For the purposes of interface matching and descriptor set `operations`, if a resource variable is not an array, it is treated as if it has an `arrayElement` of zero.
There is a limit on the number of resources of each type that can be accessed by a pipeline stage as shown in Shader Resource Limits. The “Resources Per Stage” column gives the limit on the number each type of resource that can be statically used for an entry point in any given stage in a pipeline. The “Resource Types” column lists which resource types are counted against the limit. Some resource types count against multiple limits. The VK_DESCRIPTOR_TYPE_MUTABLE_VALVE descriptor type counts as one individual resource and one for every unique resource limit per descriptor set type that is present in the associated binding’s stype:VkMutableDescriptorTypeListVALVE. If multiple descriptor types in stype:VkMutableDescriptorTypeListVALVE map to the same resource limit, only one descriptor is consumed for purposes of computing resource limits.

The pipeline layout may include descriptor sets and bindings which are not referenced by any variables statically used by the entry points for the shader stages in the binding's stageFlags.

However, if a variable assigned to a given DescriptorSet and Binding is statically used by the entry point for a shader stage, the pipeline layout must contain a descriptor set layout binding in that descriptor set layout and for that binding number, and that binding's stageFlags must include the appropriate VkShaderStageFlagBits for that stage. The variable must be of a valid resource type determined by its SPIR-V type and storage class, as defined in Shader Resource and Storage Class Correspondence. The descriptor set layout binding must be of a corresponding descriptor type, as defined in Shader Resource and Descriptor Type Correspondence.

Note

There are no limits on the number of shader variables that can have overlapping set and binding values in a shader; but which resources are statically used has an impact. If any shader variable identifying a resource is statically used in a shader, then the underlying descriptor bound at the declared set and binding must support the declared type in the shader when the shader executes.

If multiple shader variables are declared with the same set and binding values, and with the same underlying descriptor type, they can all be statically used within the same shader. However, accesses are not automatically synchronized, and Aliased decorations should be used to avoid data hazards (see section 2.18.2 Aliasing in the SPIR-V specification).

If multiple shader variables with the same set and binding values are declared in a single shader, but with different declared types, where any of those are not supported by the relevant bound descriptor, that shader can only be executed if the variables with the unsupported type are not statically used.

A noteworthy example of using multiple statically-used shader variables sharing the same descriptor set and binding values is a descriptor of type VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER that has multiple corresponding shader variables in the UniformConstant storage class, where some could be OpTypeImage (Sampled=1), some could be OpTypeSampler, and some could be OpTypeSampledImage.
## Resources per Stage

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maxPerStageDescriptorSamplers</code> or <code>maxPerStageDescriptorUpdateAfterBindSamplers</code></td>
<td>Sampler or combined image sampler</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorSampledImages</code> or <code>maxPerStageDescriptorUpdateAfterBindSampledImages</code></td>
<td>Sampled image or combined image sampler</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorStorageImages</code> or <code>maxPerStageDescriptorUpdateAfterBindStorageImages</code></td>
<td>Storage image or storage texel buffer</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorUniformBuffers</code> or <code>maxPerStageDescriptorUpdateAfterBindUniformBuffers</code></td>
<td>Uniform buffer or uniform buffer dynamic</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorStorageBuffers</code> or <code>maxPerStageDescriptorUpdateAfterBindStorageBuffers</code></td>
<td>Storage buffer or storage buffer dynamic</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorInputAttachments</code> or <code>maxPerStageDescriptorUpdateAfterBindInputAttachments</code></td>
<td>Input attachment</td>
</tr>
<tr>
<td><code>maxPerStageDescriptorInlineUniformBlocks</code> or <code>maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks</code></td>
<td>Inline uniform block</td>
</tr>
<tr>
<td><code>VkPhysicalDeviceRayTracingPropertiesNV::maxDescriptorSetAccelerationStructures</code> or <code>maxPerStageDescriptorAccelerationStructures</code> or <code>maxPerStageDescriptorUpdateAfterBindAccelerationStructures</code></td>
<td>Acceleration structure</td>
</tr>
</tbody>
</table>

---

1. Input attachments can only be used in the fragment shader stage.

### 15.6.4. Offset and Stride Assignment

Certain objects must be explicitly laid out using the Offset, ArrayStride, and MatrixStride, as described in SPIR-V explicit layout validation rules. All such layouts also must conform to the following requirements.

**Note**

The numeric order of Offset decorations does not need to follow member declaration order.

**Alignment Requirements**

There are different alignment requirements depending on the specific resources and on the features enabled on the device.
The scalar alignment of the type of an OpTypeStruct member is defined recursively as follows:

- A scalar of size N has a scalar alignment of N.
- A vector or matrix type has a scalar alignment equal to that of its component type.
- An array type has a scalar alignment equal to that of its element type.
- A structure has a scalar alignment equal to the largest scalar alignment of any of its members.

The base alignment of the type of an OpTypeStruct member is defined recursively as follows:

- A scalar has a base alignment equal to its scalar alignment.
- A two-component vector has a base alignment equal to twice its scalar alignment.
- A three- or four-component vector has a base alignment equal to four times its scalar alignment.
- An array has a base alignment equal to the base alignment of its element type.
- A structure has a base alignment equal to the largest base alignment of any of its members.
- A row-major matrix of C columns has a base alignment equal to the base alignment of a vector of C matrix components.
- A column-major matrix has a base alignment equal to the base alignment of the matrix column type.

The extended alignment of the type of an OpTypeStruct member is similarly defined as follows:

- A scalar, vector or matrix type has an extended alignment equal to its base alignment.
- An array or structure type has an extended alignment equal to the largest extended alignment of any of its members, rounded up to a multiple of 16.

A member is defined to improperly straddle if either of the following are true:

- It is a vector with total size less than or equal to 16 bytes, and has Offset decorations placing its first byte at F and its last byte at L, where floor(F / 16) ≠ floor(L / 16).
- It is a vector with total size greater than 16 bytes and has its Offset decorations placing its first byte at a non-integer multiple of 16.

Standard Buffer Layout

Every member of an OpTypeStruct that is required to be explicitly laid out must be aligned according to the first matching rule as follows. If the struct is contained in pointer types of multiple storage classes, it must satisfy the requirements for every storage class used to reference it.

1. If the scalarBlockLayout feature is enabled on the device and the storage class is Uniform, StorageBuffer, PhysicalStorageBuffer, or PushConstant then every member must be aligned according to its scalar alignment.
2. If the workgroupMemoryExplicitLayoutScalarBlockLayout feature is enabled on the device and the storage class is Workgroup then every member must be aligned according to its scalar alignment.
3. All vectors must be aligned according to their scalar alignment.
4. If the uniformBufferStandardLayout feature is not enabled on the device, then any member of an
OpTypeStruct with a storage class of Uniform and a decoration of Block must be aligned according to its extended alignment.

5. Every other member must be aligned according to its base alignment.

Note

Even if scalar alignment is supported, it is generally more performant to use the base alignment.

The memory layout must obey the following rules:

- The Offset decoration of any member must be a multiple of its alignment.
- Any ArrayStride or MatrixStride decoration must be a multiple of the alignment of the array or matrix as defined above.

If one of the conditions below applies

- The storage class is Uniform, StorageBuffer, PhysicalStorageBuffer, or PushConstant, and the scalarBlockLayout feature is not enabled on the device.
- The storage class is Workgroup, and either the struct member is not part of a Block or the workgroupMemoryExplicitLayoutScalarBlockLayout feature is not enabled on the device.
- The storage class is any other storage class.

the memory layout must also obey the following rules:

- Vectors must not improperly straddle, as defined above.
- The Offset decoration of a member must not place it between the end of a structure or an array and the next multiple of the alignment of that structure or array.

Note

The std430 layout in GLSL satisfies these rules for types using the base alignment. The std140 layout satisfies the rules for types using the extended alignment.

15.7. Built-In Variables

Built-in variables are accessed in shaders by declaring a variable decorated with a BuiltIn SPIR-V decoration. The meaning of each BuiltIn decoration is as follows. In the remainder of this section, the name of a built-in is used interchangeably with a term equivalent to a variable decorated with that particular built-in. Built-ins that represent integer values can be declared as either signed or unsigned 32-bit integers.

As mentioned above, some inputs and outputs have an additional level of arrayness relative to other shader inputs and outputs. This level of arrayness is not included in the type descriptions below, but must be included when declaring the built-in.

BaryCoordNV

The BaryCoordNV decoration can be used to decorate a fragment shader input variable. This
variable will contain a three-component floating-point vector with barycentric weights that indicate the location of the fragment relative to the screen-space locations of vertices of its primitive, obtained using perspective interpolation.

Valid Usage

- VUID-BaryCoordNV-BaryCoordNV-04154
  The BaryCoordNV decoration must be used only within the Fragment Execution Model.

- VUID-BaryCoordNV-BaryCoordNV-04155
  The variable decorated with BaryCoordNV must be declared using the Input Storage Class.

- VUID-BaryCoordNV-BaryCoordNV-04156
  The variable decorated with BaryCoordNV must be declared as a three-component vector of 32-bit floating-point values.

BaryCoordNoPerspAMD

The BaryCoordNoPerspAMD decoration can be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using linear interpolation at the fragment's center. The K coordinate of the barycentric coordinates can be derived given the identity I + J + K = 1.0.

Valid Usage

- VUID-BaryCoordNoPerspAMD-BaryCoordNoPerspAMD-04157
  The BaryCoordNoPerspAMD decoration must be used only within the Fragment Execution Model.

- VUID-BaryCoordNoPerspAMD-BaryCoordNoPerspAMD-04158
  The variable decorated with BaryCoordNoPerspAMD must be declared using the Input Storage Class.

- VUID-BaryCoordNoPerspAMD-BaryCoordNoPerspAMD-04159
  The variable decorated with BaryCoordNoPerspAMD must be declared as a two-component vector of 32-bit floating-point values.

BaryCoordNoPerspNV

The BaryCoordNoPerspNV decoration can be used to decorate a fragment shader input variable. This variable will contain a three-component floating-point vector with barycentric weights that indicate the location of the fragment relative to the screen-space locations of vertices of its primitive, obtained using linear interpolation.
Valid Usage

- VUID-BaryCoordNoPerspNV-BaryCoordNoPerspNV-04160
  The `BaryCoordNoPerspNV` decoration **must** be used only within the Fragment Execution Model.

- VUID-BaryCoordNoPerspNV-BaryCoordNoPerspNV-04161
  The variable decorated with `BaryCoordNoPerspNV` **must** be declared using the Input Storage Class.

- VUID-BaryCoordNoPerspNV-BaryCoordNoPerspNV-04162
  The variable decorated with `BaryCoordNoPerspNV` **must** be declared as a three-component vector of 32-bit floating-point values.

**BaryCoordNoPerspCentroidAMD**

The `BaryCoordNoPerspCentroidAMD` decoration **can** be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using linear interpolation at the centroid. The K coordinate of the barycentric coordinates **can** be derived given the identity I + J + K = 1.0.

Valid Usage

- VUID-BaryCoordNoPerspCentroidAMD-BaryCoordNoPerspCentroidAMD-04163
  The `BaryCoordNoPerspCentroidAMD` decoration **must** be used only within the Fragment Execution Model.

- VUID-BaryCoordNoPerspCentroidAMD-BaryCoordNoPerspCentroidAMD-04164
  The variable decorated with `BaryCoordNoPerspCentroidAMD` **must** be declared using the Input Storage Class.

- VUID-BaryCoordNoPerspCentroidAMD-BaryCoordNoPerspCentroidAMD-04165
  The variable decorated with `BaryCoordNoPerspCentroidAMD` **must** be declared as a three-component vector of 32-bit floating-point values.

**BaryCoordNoPerspSampleAMD**

The `BaryCoordNoPerspSampleAMD` decoration **can** be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using linear interpolation at each covered sample. The K coordinate of the barycentric coordinates **can** be derived given the identity I + J + K = 1.0.
Valid Usage

- VUID-BaryCoordNoPerspSampleAMD-BaryCoordNoPerspSampleAMD-04166
  The BaryCoordNoPerspSampleAMD decoration must be used only within the Fragment Execution Model.

- VUID-BaryCoordNoPerspSampleAMD-BaryCoordNoPerspSampleAMD-04167
  The variable decorated with BaryCoordNoPerspSampleAMD must be declared using the Input Storage Class.

- VUID-BaryCoordNoPerspSampleAMD-BaryCoordNoPerspSampleAMD-04168
  The variable decorated with BaryCoordNoPerspSampleAMD must be declared as a two-component vector of 32-bit floating-point values.

BaryCoordPullModelAMD

The BaryCoordPullModelAMD decoration can be used to decorate a fragment shader input variable. This variable will contain \((1/W, 1/I, 1/J)\) evaluated at the fragment center and can be used to calculate gradients and then interpolate I, J, and W at any desired sample location.

Valid Usage

- VUID-BaryCoordPullModelAMD-BaryCoordPullModelAMD-04169
  The BaryCoordPullModelAMD decoration must be used only within the Fragment Execution Model.

- VUID-BaryCoordPullModelAMD-BaryCoordPullModelAMD-04170
  The variable decorated with BaryCoordPullModelAMD must be declared using the Input Storage Class.

- VUID-BaryCoordPullModelAMD-BaryCoordPullModelAMD-04171
  The variable decorated with BaryCoordPullModelAMD must be declared as a three-component vector of 32-bit floating-point values.

BaryCoordSmoothAMD

The BaryCoordSmoothAMD decoration can be used to decorate a fragment shader input variable. This variable will contain the \((I,J)\) pair of the barycentric coordinates corresponding to the fragment evaluated using perspective interpolation at the fragment's center. The K coordinate of the barycentric coordinates can be derived given the identity \(I + J + K = 1.0\).
Valid Usage

- VUID-BaryCoordSmoothAMD-BaryCoordSmoothAMD-04172
  The BaryCoordSmoothAMD decoration must be used only within the Fragment Execution Model

- VUID-BaryCoordSmoothAMD-BaryCoordSmoothAMD-04173
  The variable decorated with BaryCoordSmoothAMD must be declared using the Input Storage Class

- VUID-BaryCoordSmoothAMD-BaryCoordSmoothAMD-04174
  The variable decorated with BaryCoordSmoothAMD must be declared as a two-component vector of 32-bit floating-point values

BaryCoordSmoothCentroidAMD

The BaryCoordSmoothCentroidAMD decoration can be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using perspective interpolation at the centroid. The K coordinate of the barycentric coordinates can be derived given the identity I + J + K = 1.0.

Valid Usage

- VUID-BaryCoordSmoothCentroidAMD-BaryCoordSmoothCentroidAMD-04175
  The BaryCoordSmoothCentroidAMD decoration must be used only within the Fragment Execution Model

- VUID-BaryCoordSmoothCentroidAMD-BaryCoordSmoothCentroidAMD-04176
  The variable decorated with BaryCoordSmoothCentroidAMD must be declared using the Input Storage Class

- VUID-BaryCoordSmoothCentroidAMD-BaryCoordSmoothCentroidAMD-04177
  The variable decorated with BaryCoordSmoothCentroidAMD must be declared as a two-component vector of 32-bit floating-point values

BaryCoordSmoothSampleAMD

The BaryCoordSmoothSampleAMD decoration can be used to decorate a fragment shader input variable. This variable will contain the (I,J) pair of the barycentric coordinates corresponding to the fragment evaluated using perspective interpolation at each covered sample. The K coordinate of the barycentric coordinates can be derived given the identity I + J + K = 1.0.
Valid Usage

• VUID-BaryCoordSmoothSampleAMD-BaryCoordSmoothSampleAMD-04178
  The BaryCoordSmoothSampleAMD decoration must be used only within the Fragment Execution Model

• VUID-BaryCoordSmoothSampleAMD-BaryCoordSmoothSampleAMD-04179
  The variable decorated with BaryCoordSmoothSampleAMD must be declared using the Input Storage Class

• VUID-BaryCoordSmoothSampleAMD-BaryCoordSmoothSampleAMD-04180
  The variable decorated with BaryCoordSmoothSampleAMD must be declared as a two-component vector of 32-bit floating-point values

BaseInstance

Decorating a variable with the BaseInstance built-in will make that variable contain the integer value corresponding to the first instance that was passed to the command that invoked the current vertex shader invocation. BaseInstance is the firstInstance parameter to a direct drawing command or the firstInstance member of a structure consumed by an indirect drawing command.

Valid Usage

• VUID-BaseInstance-BaseInstance-04181
  The BaseInstance decoration must be used only within the Vertex Execution Model

• VUID-BaseInstance-BaseInstance-04182
  The variable decorated with BaseInstance must be declared using the Input Storage Class

• VUID-BaseInstance-BaseInstance-04183
  The variable decorated with BaseInstance must be declared as a scalar 32-bit integer value

BaseVertex

Decorating a variable with the BaseVertex built-in will make that variable contain the integer value corresponding to the first vertex or vertex offset that was passed to the command that invoked the current vertex shader invocation. For non-indexed drawing commands, this variable is the firstVertex parameter to a direct drawing command or the firstVertex member of the structure consumed by an indirect drawing command. For indexed drawing commands, this variable is the vertexOffset parameter to a direct drawing command or the vertexOffset member of the structure consumed by an indirect drawing command.
Valid Usage

- VUID-BaseVertex-BaseVertex-04184
  The `BaseVertex` decoration **must** be used only within the Vertex Execution Model

- VUID-BaseVertex-BaseVertex-04185
  The variable decorated with `BaseVertex` **must** be declared using the Input Storage Class

- VUID-BaseVertex-BaseVertex-04186
  The variable decorated with `BaseVertex` **must** be declared as a scalar 32-bit integer value

**ClipDistance**

Decorating a variable with the `ClipDistance` built-in decoration will make that variable contain the mechanism for controlling user clipping. `ClipDistance` is an array such that the \( i \)th element of the array specifies the clip distance for plane \( i \). A clip distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip half-space, and a negative distance means the vertex is outside the clip half-space.

**Note**
The array variable decorated with `ClipDistance` is explicitly sized by the shader.

**Note**
In the last vertex processing stage, these values will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances less than 0 will be considered outside the clip volume. If `ClipDistance` is then used by a fragment shader, `ClipDistance` contains these linearly interpolated values.
Valid Usage

- **VUID-ClipDistance-ClipDistance-04187**
  The ClipDistance decoration must be used only within the MeshNV, Vertex, Fragment, TessellationControl, TessellationEvaluation, or Geometry Execution Model.

- **VUID-ClipDistance-ClipDistance-04188**
  The variable decorated with ClipDistance within the MeshNV or Vertex Execution Model must be declared using the Output Storage Class.

- **VUID-ClipDistance-ClipDistance-04189**
  The variable decorated with ClipDistance within the Fragment Execution Model must be declared using the Input Storage Class.

- **VUID-ClipDistance-ClipDistance-04190**
  The variable decorated with ClipDistance within the TessellationControl, TessellationEvaluation, or Geometry Execution Model must not be declared in a Storage Class other than Input or Output.

- **VUID-ClipDistance-ClipDistance-04191**
  The variable decorated with ClipDistance must be declared as an array of 32-bit floating-point values.

**ClipDistancePerViewNV**

Decorating a variable with the ClipDistancePerViewNV built-in decoration will make that variable contain the per-view clip distances. The per-view clip distances have the same semantics as ClipDistance.

Valid Usage

- **VUID-ClipDistancePerViewNV-ClipDistancePerViewNV-04192**
  The ClipDistancePerViewNV decoration must be used only within the MeshNV Execution Model.

- **VUID-ClipDistancePerViewNV-ClipDistancePerViewNV-04193**
  The variable decorated with ClipDistancePerViewNV must be declared using the Output Storage Class.

- **VUID-ClipDistancePerViewNV-ClipDistancePerViewNV-04194**
  The variable decorated with ClipDistancePerViewNV must also be decorated with the PerViewNV decoration.

- **VUID-ClipDistancePerViewNV-ClipDistancePerViewNV-04195**
  The variable decorated with ClipDistancePerViewNV must be declared as a two-dimensional array of 32-bit floating-point values.

**CullDistance**

Decorating a variable with the CullDistance built-in decoration will make that variable contain the mechanism for controlling user culling. If any member of this array is assigned a negative value for all vertices belonging to a primitive, then the primitive is discarded before rasterization.
Note
In fragment shaders, the values of the `CullDistance` array are linearly interpolated across each primitive.

Note
If `CullDistance` decorates an input variable, that variable will contain the corresponding value from the `CullDistance` decorated output variable from the previous shader stage.

Valid Usage

- **VUID-CullDistance-CullDistance-04196**
  The `CullDistance` decoration must be used only within the MeshNV, Vertex, Fragment, TessellationControl, TessellationEvaluation, or Geometry Execution Model

- **VUID-CullDistance-CullDistance-04197**
  The variable decorated with `CullDistance` within the MeshNV or Vertex Execution Model must be declared using the Output Storage Class

- **VUID-CullDistance-CullDistance-04198**
  The variable decorated with `CullDistance` within the Fragment Execution Model must be declared using the Input Storage Class

- **VUID-CullDistance-CullDistance-04199**
  The variable decorated with `CullDistance` within the TessellationControl, TessellationEvaluation, or Geometry Execution Model must not be declared using a Storage Class other than Input or Output

- **VUID-CullDistance-CullDistance-04200**
  The variable decorated with `CullDistance` must be declared as an array of 32-bit floating-point values

`CullDistancePerViewNV`

Decorating a variable with the `CullDistancePerViewNV` built-in decoration will make that variable contain the per-view cull distances. The per-view clip distances have the same semantics as `CullDistance`. 
Valid Usage

- VUID-CullDistancePerViewNV-CullDistancePerViewNV-04201
  The `CullDistancePerViewNV` decoration **must** be used only within the `MeshNV Execution Model`.

- VUID-CullDistancePerViewNV-CullDistancePerViewNV-04202
  The variable decorated with `CullDistancePerViewNV` **must** be declared using the `Output Storage Class`.

- VUID-CullDistancePerViewNV-CullDistancePerViewNV-04203
  The variable decorated with `CullDistancePerViewNV` **must** also be decorated with the `PerViewNV` decoration.

- VUID-CullDistancePerViewNV-CullDistancePerViewNV-04204
  The variable decorated with `CullDistancePerViewNV` **must** be declared as a two-dimensional array of 32-bit floating-point values.

---

DeviceIndex

The `DeviceIndex` decoration **can** be applied to a shader input which will be filled with the device index of the physical device that is executing the current shader invocation. This value will be in the range `{0, max(1, physicalDeviceCount)}`, where `physicalDeviceCount` is the `physicalDeviceCount` member of `VkDeviceGroupDeviceCreateInfo`.

Valid Usage

- VUID-DeviceIndex-DeviceIndex-04205
  The variable decorated with `DeviceIndex` **must** be declared using the `Input Storage Class`.

- VUID-DeviceIndex-DeviceIndex-04206
  The variable decorated with `DeviceIndex` **must** be declared as a scalar 32-bit integer value.

---

DrawIndex

Decorating a variable with the `DrawIndex` built-in will make that variable contain the integer value corresponding to the zero-based index of the drawing command that invoked the current task, mesh, or vertex shader invocation. For *indirect drawing commands*, `DrawIndex` begins at zero and increments by one for each draw command executed. The number of draw commands is given by the `drawCount` parameter. For *direct drawing commands*, `DrawIndex` is always zero. `DrawIndex` is dynamically uniform.

When task or mesh shaders are used, only the first active stage will have proper access to the variable. The value read by other stages is undefined.
Valid Usage

- VUID-DrawIndex-DrawIndex-04207
  The `DrawIndex` decoration **must** be used only within the `Vertex`, `MeshNV`, or `TaskNV` Execution Model

- VUID-DrawIndex-DrawIndex-04208
  The variable decorated with `DrawIndex` **must** be declared using the `Input Storage Class`

- VUID-DrawIndex-DrawIndex-04209
  The variable decorated with `DrawIndex` **must** be declared as a scalar 32-bit integer value

FragCoord

Decorating a variable with the `FragCoord` built-in decoration will make that variable contain the framebuffer coordinate \((x, y, z, \frac{1}{w})\) of the fragment being processed. The \((x,y)\) coordinate \((0,0)\) is the upper left corner of the upper left pixel in the framebuffer.

When **Sample Shading** is enabled, the \(x\) and \(y\) components of `FragCoord` reflect the location of one of the samples corresponding to the shader invocation.

Otherwise, the \(x\) and \(y\) components of `FragCoord` reflect the location of the center of the fragment.

The \(z\) component of `FragCoord` is the interpolated depth value of the primitive.

The \(w\) component is the interpolated \(\frac{1}{w}\).

The **Centroid** interpolation decoration is ignored, but allowed, on `FragCoord`.

Valid Usage

- VUID-FragCoord-FragCoord-04210
  The `FragCoord` decoration **must** be used only within the Fragment Execution Model

- VUID-FragCoord-FragCoord-04211
  The variable decorated with `FragCoord` **must** be declared using the `Input Storage Class`

- VUID-FragCoord-FragCoord-04212
  The variable decorated with `FragCoord` **must** be declared as a four-component vector of 32-bit floating-point values

FragDepth

To have a shader supply a fragment-depth value, the shader **must** declare the `DepthReplacing` execution mode. Such a shader’s fragment-depth value will come from the variable decorated with the `FragDepth` built-in decoration.

This value will be used for any subsequent depth testing performed by the implementation or writes to the depth attachment.
Valid Usage

• VUID-FragDepth-FragDepth-04213
  The **FragDepth** decoration **must** be used only within the **Fragment Execution Model**

• VUID-FragDepth-FragDepth-04214
  The variable decorated with **FragDepth** **must** be declared using the **Output Storage Class**

• VUID-FragDepth-FragDepth-04215
  The variable decorated with **FragDepth** **must** be declared as a scalar 32-bit floating-point value

• VUID-FragDepth-FragDepth-04216
  If the shader dynamically writes to the variable decorated with **FragDepth**, the **DepthReplacing Execution Mode** **must** be declared

**FragInvocationCountEXT**

Decorating a variable with the **FragInvocationCountEXT** built-in decoration will make that variable contain the maximum number of fragment shader invocations for the fragment, as determined by **minSampleShading**.

If **Sample Shading** is not enabled, **FragInvocationCountEXT** will be filled with a value of 1.

Valid Usage

• VUID-FragInvocationCountEXT-FragInvocationCountEXT-04217
  The **FragInvocationCountEXT** decoration **must** be used only within the **Fragment Execution Model**

• VUID-FragInvocationCountEXT-FragInvocationCountEXT-04218
  The variable decorated with **FragInvocationCountEXT** **must** be declared using the **Input Storage Class**

• VUID-FragInvocationCountEXT-FragInvocationCountEXT-04219
  The variable decorated with **FragInvocationCountEXT** **must** be declared as a scalar 32-bit integer value

**FragSizeEXT**

Decorating a variable with the **FragSizeEXT** built-in decoration will make that variable contain the dimensions in pixels of the area that the fragment covers for that invocation.

If fragment density map is not enabled, **FragSizeEXT** will be filled with a value of (1,1).
Valid Usage

- VUID-FragSizeEXT-FragSizeEXT-04220
  The `FragSizeEXT` decoration must be used only within the Fragment Execution Model

- VUID-FragSizeEXT-FragSizeEXT-04221
  The variable decorated with `FragSizeEXT` must be declared using the Input Storage Class

- VUID-FragSizeEXT-FragSizeEXT-04222
  The variable decorated with `FragSizeEXT` must be declared as a two-component vector of 32-bit integer values

`FragStencilRefEXT`

Decorating a variable with the `FragStencilRefEXT` built-in decoration will make that variable contain the new stencil reference value for all samples covered by the fragment. This value will be used as the stencil reference value used in stencil testing.

To write to `FragStencilRefEXT`, a shader must declare the `StencilRefReplacingEXT` execution mode. If a shader declares the `StencilRefReplacingEXT` execution mode and there is an execution path through the shader that does not set `FragStencilRefEXT`, then the fragment’s stencil reference value is undefined for executions of the shader that take that path.

Only the least significant $s$ bits of the integer value of the variable decorated with `FragStencilRefEXT` are considered for stencil testing, where $s$ is the number of bits in the stencil framebuffer attachment, and higher order bits are discarded.

Valid Usage

- VUID-FragStencilRefEXT-FragStencilRefEXT-04223
  The `FragStencilRefEXT` decoration must be used only within the Fragment Execution Model

- VUID-FragStencilRefEXT-FragStencilRefEXT-04224
  The variable decorated with `FragStencilRefEXT` must be declared using the Output Storage Class

- VUID-FragStencilRefEXT-FragStencilRefEXT-04225
  The variable decorated with `FragStencilRefEXT` must be declared as a scalar integer value

`FragmentSizeNV`

Decorating a variable with the `FragmentSizeNV` built-in decoration will make that variable contain the width and height of the fragment.
Valid Usage

- **VUID-FragmentSizeNV-FragmentSizeNV-04226**
  The *FragmentSizeNV* decoration must be used only within the *Fragment Execution Model*

- **VUID-FragmentSizeNV-FragmentSizeNV-04227**
  The variable decorated with *FragmentSizeNV* must be declared using the *Input Storage Class*

- **VUID-FragmentSizeNV-FragmentSizeNV-04228**
  The variable decorated with *FragmentSizeNV* must be declared as a two-component vector of 32-bit integer values

**FrontFacing**

Decorating a variable with the *FrontFacing* built-in decoration will make that variable contain whether the fragment is front or back facing. This variable is non-zero if the current fragment is considered to be part of a *front-facing* polygon primitive or of a non-polygon primitive and is zero if the fragment is considered to be part of a back-facing polygon primitive.

Valid Usage

- **VUID-FrontFacing-FrontFacing-04229**
  The *FrontFacing* decoration must be used only within the *Fragment Execution Model*

- **VUID-FrontFacing-FrontFacing-04230**
  The variable decorated with *FrontFacing* must be declared using the *Input Storage Class*

- **VUID-FrontFacing-FrontFacing-04231**
  The variable decorated with *FrontFacing* must be declared as a boolean value

**FullyCoveredEXT**

Decorating a variable with the *FullyCoveredEXT* built-in decoration will make that variable indicate whether the *fragment area* is fully covered by the generating primitive. This variable is non-zero if conservative rasterization is enabled and the current fragment area is fully covered by the generating primitive, and is zero if the fragment is not covered or partially covered, or conservative rasterization is disabled.

If the implementation supports *VkPhysicalDeviceConservativeRasterizationPropertiesEXT*::*conservativeRasterizationPostDepthCoverage* and the *PostDepthCoverage* execution mode is specified the *SampleMask* built-in input variable will reflect the coverage after the early post-fraction depth and stencil tests are applied.
Valid Usage

- VUID-FullyCoveredEXT-FullyCoveredEXT-04232
  The **FullyCoveredEXT** decoration **must** be used only within the Fragment Execution Model

- VUID-FullyCoveredEXT-FullyCoveredEXT-04233
  The variable decorated with **FullyCoveredEXT** **must** be declared using the Input Storage Class

- VUID-FullyCoveredEXT-FullyCoveredEXT-04234
  The variable decorated with **FullyCoveredEXT** **must** be declared as a boolean value

- VUID-FullyCoveredEXT-conservativeRasterizationPostDepthCoverage-04235
  If `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` is not supported the PostDepthCoverage Execution Mode **must** not be declared, when a variable with the **FullyCoveredEXT** decoration is declared

GlobalInvocationId

Decorating a variable with the **GlobalInvocationId** built-in decoration will make that variable contain the location of the current invocation within the global workgroup. Each component is equal to the index of the local workgroup multiplied by the size of the local workgroup plus **LocalInvocationId**.

Valid Usage

- VUID-GlobalInvocationId-GlobalInvocationId-04236
  The **GlobalInvocationId** decoration **must** be used only within the GLCompute, MeshNV, or TaskNV Execution Model

- VUID-GlobalInvocationId-GlobalInvocationId-04237
  The variable decorated with **GlobalInvocationId** **must** be declared using the Input Storage Class

- VUID-GlobalInvocationId-GlobalInvocationId-04238
  The variable decorated with **GlobalInvocationId** **must** be declared as a three-component vector of 32-bit integer values

HelperInvocation

Decorating a variable with the **HelperInvocation** built-in decoration will make that variable contain whether the current invocation is a helper invocation. This variable is non-zero if the current fragment being shaded is a helper invocation and zero otherwise. A helper invocation is an invocation of the shader that is produced to satisfy internal requirements such as the generation of derivatives.

**Note**

It is very likely that a helper invocation will have a value of **SampleMask** fragment shader input value that is zero.
Valid Usage

- VUID-HelperInvocation-HelperInvocation-04239
  The **HelperInvocation** decoration **must** be used only within the **Fragment Execution Model**

- VUID-HelperInvocation-HelperInvocation-04240
  The variable decorated with **HelperInvocation** must be declared using the **Input Storage Class**

- VUID-HelperInvocation-HelperInvocation-04241
  The variable decorated with **HelperInvocation** must be declared as a boolean value

**HitKindKHR**

A variable decorated with the **HitKindKHR** decoration will describe the intersection that triggered the execution of the current shader. The values are determined by the intersection shader. For user-defined intersection shaders this is the value that was passed to the “Hit Kind” operand of **OpReportIntersectionKHR**. For triangle intersection candidates, this will be one of **HitKindFrontFacingTriangleKHR** or **HitKindBackFacingTriangleKHR**.

Valid Usage

- VUID-HitKindKHR-HitKindKHR-04242
  The **HitKindKHR** decoration **must** be used only within the **AnyHitKHR** or **ClosestHitKHR** Execution Model

- VUID-HitKindKHR-HitKindKHR-04243
  The variable decorated with **HitKindKHR** must be declared using the **Input Storage Class**

- VUID-HitKindKHR-HitKindKHR-04244
  The variable decorated with **HitKindKHR** must be declared as a scalar 32-bit integer value

**HitTNV**

A variable decorated with the **HitTNV** decoration is equivalent to a variable decorated with the **RayTmaxKHR** decoration.

Valid Usage

- VUID-HitTNV-HitTNV-04245
  The **HitTNV** decoration **must** be used only within the **AnyHitNV** or **ClosestHitNV** Execution Model

- VUID-HitTNV-HitTNV-04246
  The variable decorated with **HitTNV** must be declared using the **Input Storage Class**

- VUID-HitTNV-HitTNV-04247
  The variable decorated with **HitTNV** must be declared as a scalar 32-bit floating-point value
IncomingRayFlagsKHR

A variable with the `IncomingRayFlagsKHR` decoration will contain the ray flags passed in to the trace call that invoked this particular shader. Setting pipeline flags on the raytracing pipeline must not cause any corresponding flags to be set in variables with this decoration.

Valid Usage

- VUID-IncomingRayFlagsKHR-IncomingRayFlagsKHR-04248
  The `IncomingRayFlagsKHR` decoration must be used only within the `IntersectionKHR`, `AnyHitKHR`, `ClosestHitKHR`, or `MissKHR` Execution Model

- VUID-IncomingRayFlagsKHR-IncomingRayFlagsKHR-04249
  The variable decorated with `IncomingRayFlagsKHR` must be declared using the Input Storage Class

- VUID-IncomingRayFlagsKHR-IncomingRayFlagsKHR-04250
  The variable decorated with `IncomingRayFlagsKHR` must be declared as a scalar 32-bit integer value

InstanceCustomIndexKHR

A variable decorated with the `InstanceCustomIndexKHR` decoration will contain the application-defined value of the instance that intersects the current ray. This variable contains the value that was specified in `VkAccelerationStructureInstanceKHR::instanceCustomIndex` for the current acceleration structure instance in the lower 24 bits and the upper 8 bits will be zero.

Valid Usage

- VUID-InstanceCustomIndexKHR-InstanceCustomIndexKHR-04251
  The `InstanceCustomIndexKHR` decoration must be used only within the `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR` Execution Model

- VUID-InstanceCustomIndexKHR-InstanceCustomIndexKHR-04252
  The variable decorated with `InstanceCustomIndexKHR` must be declared using the Input Storage Class

- VUID-InstanceCustomIndexKHR-InstanceCustomIndexKHR-04253
  The variable decorated with `InstanceCustomIndexKHR` must be declared as a scalar 32-bit integer value

InstanceId

Decorating a variable in an intersection, any-hit, or closest hit shader with the `InstanceId` decoration will make that variable contain the index of the instance that intersects the current ray.
Valid Usage

- VUID-InstanceId-InstanceId-04254
  The `InstanceId` decoration must be used only within the `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR` Execution Model

- VUID-InstanceId-InstanceId-04255
  The variable decorated with `InstanceId` must be declared using the `Input Storage Class`

- VUID-InstanceId-InstanceId-04256
  The variable decorated with `InstanceId` must be declared as a scalar 32-bit integer value

InvocationId

Decorating a variable with the `InvocationId` built-in decoration will make that variable contain the index of the current shader invocation in a geometry shader, or the index of the output patch vertex in a tessellation control shader.

In a geometry shader, the index of the current shader invocation ranges from zero to the number of instances declared in the shader minus one. If the instance count of the geometry shader is one or is not specified, then `InvocationId` will be zero.

Valid Usage

- VUID-InvocationId-InvocationId-04257
  The `InvocationId` decoration must be used only within the `TessellationControl` or `Geometry Execution Model`

- VUID-InvocationId-InvocationId-04258
  The variable decorated with `InvocationId` must be declared using the `Input Storage Class`

- VUID-InvocationId-InvocationId-04259
  The variable decorated with `InvocationId` must be declared as a scalar 32-bit integer value

InvocationsPerPixelNV

Decorating a variable with the `InvocationsPerPixelNV` built-in decoration will make that variable contain the maximum number of fragment shader invocations per pixel, as derived from the effective shading rate for the fragment. If a primitive does not fully cover a pixel, the number of fragment shader invocations for that pixel may be less than the value of `InvocationsPerPixelNV`. If the shading rate indicates a fragment covering multiple pixels, then `InvocationsPerPixelNV` will be one.
Valid Usage

- VUID-InvocationsPerPixelNV-InvocationsPerPixelNV-04260
  The InvocationsPerPixelNV decoration must be used only within the Fragment Execution Model

- VUID-InvocationsPerPixelNV-InvocationsPerPixelNV-04261
  The variable decorated with InvocationsPerPixelNV must be declared using the Input Storage Class

- VUID-InvocationsPerPixelNV-InvocationsPerPixelNV-04262
  The variable decorated with InvocationsPerPixelNV must be declared as a scalar 32-bit integer value

InstanceIndex

Decorating a variable in a vertex shader with the InstanceIndex built-in decoration will make that variable contain the index of the instance that is being processed by the current vertex shader invocation. InstanceIndex begins at the firstInstance parameter to vkCmdDraw or vkCmdDrawIndexed or at the firstInstance member of a structure consumed by vkCmdDrawIndirect or vkCmdDrawIndexedIndirect.

Valid Usage

- VUID-InstanceIndex-InstanceIndex-04263
  The InstanceIndex decoration must be used only within the Vertex Execution Model

- VUID-InstanceIndex-InstanceIndex-04264
  The variable decorated with InstanceIndex must be declared using the Input Storage Class

- VUID-InstanceIndex-InstanceIndex-04265
  The variable decorated with InstanceIndex must be declared as a scalar 32-bit integer value

LaunchIdKHR

A variable decorated with the LaunchIdKHR decoration will specify the index of the work item being processed. One work item is generated for each of the width × height × depth items dispatched by a vkCmdTraceRaysKHR command. All shader invocations inherit the same value for variables decorated with LaunchIdKHR.
Valid Usage

- **VUID-LaunchIdKHR-LaunchIdKHR-04266**
  The *LaunchIdKHR* decoration must be used only within the *RayGenerationKHR*, *IntersectionKHR*, *AnyHitKHR*, *ClosestHitKHR*, *MissKHR*, or *CallableKHR* Execution Model

- **VUID-LaunchIdKHR-LaunchIdKHR-04267**
  The variable decorated with *LaunchIdKHR* must be declared using the *Input Storage Class*

- **VUID-LaunchIdKHR-LaunchIdKHR-04268**
  The variable decorated with *LaunchIdKHR* must be declared as a three-component vector of 32-bit integer values

---

**LaunchSizeKHR**

A variable decorated with the *LaunchSizeKHR* decoration will contain the *width*, *height*, and *depth* dimensions passed to the *vkCmdTraceRaysKHR* command that initiated this shader execution. The *width* is in the first component, the *height* is in the second component, and the *depth* is in the third component.

Valid Usage

- **VUID-LaunchSizeKHR-LaunchSizeKHR-04269**
  The *LaunchSizeKHR* decoration must be used only within the *RayGenerationKHR*, *IntersectionKHR*, *AnyHitKHR*, *ClosestHitKHR*, *MissKHR*, or *CallableKHR* Execution Model

- **VUID-LaunchSizeKHR-LaunchSizeKHR-04270**
  The variable decorated with *LaunchSizeKHR* must be declared using the *Input Storage Class*

- **VUID-LaunchSizeKHR-LaunchSizeKHR-04271**
  The variable decorated with *LaunchSizeKHR* must be declared as a three-component vector of 32-bit integer values

---

**Layer**

Decorating a variable with the *Layer* built-in decoration will make that variable contain the select layer of a multi-layer framebuffer attachment.

In a mesh, vertex, tessellation evaluation, or geometry shader, any variable decorated with *Layer* can be written with the framebuffer layer index to which the primitive produced by that shader will be directed.

The last active *vertex processing stage* (in pipeline order) controls the *Layer* that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the *Layer*.

If the last active vertex processing stage shader entry point's interface does not include a variable decorated with *Layer*, then the first layer is used. If a vertex processing stage shader entry point's interface includes a variable decorated with *Layer*, it must write the same value to *Layer* for all output vertices of a given primitive. If the *Layer* value is less than 0 or greater than or equal to the number of layers in the framebuffer, then primitives may still be rasterized,
fragment shaders may be executed, and the framebuffer values for all layers are undefined.

If a variable with the Layer decoration is also decorated with ViewportRelativeNV, then the ViewportIndex is added to the layer that is used for rendering and that is made available in the fragment shader. If the shader writes to a variable decorated ViewportMaskNV, then the layer selected has a different value for each viewport a primitive is rendered to.

In a fragment shader, a variable decorated with Layer contains the layer index of the primitive that the fragment invocation belongs to.

Valid Usage

- VUID-Layer-Layer-04272
  The Layer decoration must be used only within the MeshNV, Vertex, TessellationEvaluation, Geometry, or Fragment Execution Model

- VUID-Layer-Layer-04274
  The variable decorated with Layer within the MeshNV, Vertex, TessellationEvaluation, or Geometry Execution Model must be declared using the Output Storage Class

- VUID-Layer-Layer-04275
  The variable decorated with Layer within the Fragment Execution Model must be declared using the Input Storage Class

- VUID-Layer-Layer-04276
  The variable decorated with Layer must be declared as a scalar 32-bit integer value

LayerPerViewNV

Decorating a variable with the LayerPerViewNV built-in decoration will make that variable contain the per-view layer information. The per-view layer has the same semantics as Layer, for each view.

Valid Usage

- VUID-LayerPerViewNV-LayerPerViewNV-04277
  The LayerPerViewNV decoration must be used only within the MeshNV Execution Model

- VUID-LayerPerViewNV-LayerPerViewNV-04278
  The variable decorated with LayerPerViewNV must be declared using the Output Storage Class

- VUID-LayerPerViewNV-LayerPerViewNV-04279
  The variable decorated with LayerPerViewNV must also be decorated with the PerViewNV decoration.

- VUID-LayerPerViewNV-LayerPerViewNV-04280
  The variable decorated with LayerPerViewNV must be declared as an array of scalar 32-bit integer values
LocalInvocationId

Decorating a variable with the `LocalInvocationId` built-in decoration will make that variable contain the location of the current task, mesh, or compute shader invocation within the local workgroup. Each component ranges from zero through to the size of the workgroup in that dimension minus one.

**Note**

If the size of the workgroup in a particular dimension is one, then the `LocalInvocationId` in that dimension will be zero. If the workgroup is effectively two-dimensional, then `LocalInvocationId.z` will be zero. If the workgroup is effectively one-dimensional, then both `LocalInvocationId.y` and `LocalInvocationId.z` will be zero.

Valid Usage

- **VUID-LocalInvocationId-LocalInvocationId-04281**
  The `LocalInvocationId` decoration **must** be used only within the GLCompute, MeshNV, or TaskNV Execution Model

- **VUID-LocalInvocationId-LocalInvocationId-04282**
  The variable decorated with `LocalInvocationId` **must** be declared using the Input Storage Class

- **VUID-LocalInvocationId-LocalInvocationId-04283**
  The variable decorated with `LocalInvocationId` **must** be declared as a three-component vector of 32-bit integer values

LocalInvocationIndex

Decorating a variable with the `LocalInvocationIndex` built-in decoration will make that variable contain a one-dimensional representation of `LocalInvocationId`. This is computed as:

```plaintext
LocalInvocationIndex =
LocalInvocationId.z * WorkgroupSize.x * WorkgroupSize.y +
LocalInvocationId.y * WorkgroupSize.x +
LocalInvocationId.x;
```
Valid Usage

- VUID-LocalInvocationIndex-LocalInvocationIndex-04284
  The `LocalInvocationIndex` decoration must be used only within the `GLCompute`, `MeshNV`, or `TaskNV` Execution Model

- VUID-LocalInvocationIndex-LocalInvocationIndex-04285
  The variable decorated with `LocalInvocationIndex` must be declared using the Input Storage Class

- VUID-LocalInvocationIndex-LocalInvocationIndex-04286
  The variable decorated with `LocalInvocationIndex` must be declared as a scalar 32-bit integer value

MeshViewCountNV

Decorating a variable with the `MeshViewCountNV` built-in decoration will make that variable contain the number of views processed by the current mesh or task shader invocations.

Valid Usage

- VUID-MeshViewCountNV-MeshViewCountNV-04287
  The `MeshViewCountNV` decoration must be used only within the `MeshNV` or `TaskNV` Execution Model

- VUID-MeshViewCountNV-MeshViewCountNV-04288
  The variable decorated with `MeshViewCountNV` must be declared using the Input Storage Class

- VUID-MeshViewCountNV-MeshViewCountNV-04289
  The variable decorated with `MeshViewCountNV` must be declared as a scalar 32-bit integer value

MeshViewIndicesNV

Decorating a variable with the `MeshViewIndicesNV` built-in decoration will make that variable contain the mesh view indices. The mesh view indices is an array of values where each element holds the view number of one of the views being processed by the current mesh or task shader invocations. The values of array elements with indices greater than or equal to `MeshViewCountNV` are undefined. If the value of `MeshViewIndicesNV[i]` is j, then any outputs decorated with `PerViewNV` will take on the value of array element i when processing primitives for view index j.
### Valid Usage

- **VUID-MeshViewIndicesNV-MeshViewIndicesNV-04290**
  The *MeshViewIndicesNV* decoration **must** be used only within the *MeshNV* or *TaskNV Execution Model*

- **VUID-MeshViewIndicesNV-MeshViewIndicesNV-04291**
  The variable decorated with *MeshViewIndicesNV* **must** be declared using the *Input Storage Class*

- **VUID-MeshViewIndicesNV-MeshViewIndicesNV-04292**
  The variable decorated with *MeshViewIndicesNV* **must** be declared as an array of scalar 32-bit integer values

#### NumSubgroups

Decorating a variable with the *NumSubgroups* built-in decoration will make that variable contain the number of subgroups in the local workgroup.

<table>
<thead>
<tr>
<th>VUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumSubgroups</td>
<td><strong>must</strong> be used only within the <em>GLCompute</em>, <em>MeshNV</em>, or <em>TaskNV Execution Model</em></td>
</tr>
<tr>
<td></td>
<td><strong>must</strong> be declared using the <em>Input Storage Class</em></td>
</tr>
<tr>
<td></td>
<td><strong>must</strong> be declared as a scalar 32-bit integer value</td>
</tr>
</tbody>
</table>

### Valid Usage

- **VUID-NumSubgroups-NumSubgroups-04293**
  The *NumSubgroups* decoration **must** be used only within the *GLCompute*, *MeshNV*, or *TaskNV Execution Model*

- **VUID-NumSubgroups-NumSubgroups-04294**
  The variable decorated with *NumSubgroups* **must** be declared using the *Input Storage Class*

- **VUID-NumSubgroups-NumSubgroups-04295**
  The variable decorated with *NumSubgroups* **must** be declared as a scalar 32-bit integer value

#### NumWorkgroups

Decorating a variable with the *NumWorkgroups* built-in decoration will make that variable contain the number of local workgroups that are part of the dispatch that the invocation belongs to. Each component is equal to the values of the workgroup count parameters passed into the dispatch commands.

<table>
<thead>
<tr>
<th>VUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumWorkgroups</td>
<td><strong>must</strong> be used only within the <em>GLCompute</em> Execution Model</td>
</tr>
<tr>
<td></td>
<td><strong>must</strong> be declared using the <em>Input Storage Class</em></td>
</tr>
<tr>
<td></td>
<td><strong>must</strong> be declared as a three-component vector of 32-bit integer values</td>
</tr>
</tbody>
</table>

---

Chapter 15. Shader Interfaces | 1043
**ObjectRayDirectionKHR**

A variable decorated with the `ObjectRayDirectionKHR` decoration will specify the direction of the ray being processed, in object space.

**Valid Usage**

- VUID-ObjectRayDirectionKHR-ObjectRayDirectionKHR-04299
  The `ObjectRayDirectionKHR` decoration must be used only within the `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR` Execution Model

- VUID-ObjectRayDirectionKHR-ObjectRayDirectionKHR-04300
  The variable decorated with `ObjectRayDirectionKHR` must be declared using the `Input Storage Class`

- VUID-ObjectRayDirectionKHR-ObjectRayDirectionKHR-04301
  The variable decorated with `ObjectRayDirectionKHR` must be declared as a three-component vector of 32-bit floating-point values

**ObjectRayOriginKHR**

A variable decorated with the `ObjectRayOriginKHR` decoration will specify the origin of the ray being processed, in object space.

**Valid Usage**

- VUID-ObjectRayOriginKHR-ObjectRayOriginKHR-04302
  The `ObjectRayOriginKHR` decoration must be used only within the `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR` Execution Model

- VUID-ObjectRayOriginKHR-ObjectRayOriginKHR-04303
  The variable decorated with `ObjectRayOriginKHR` must be declared using the `Input Storage Class`

- VUID-ObjectRayOriginKHR-ObjectRayOriginKHR-04304
  The variable decorated with `ObjectRayOriginKHR` must be declared as a three-component vector of 32-bit floating-point values

**ObjectToWorldKHR**

A variable decorated with the `ObjectToWorldKHR` decoration will contain the current object-to-world transformation matrix, which is determined by the instance of the current intersection.
**Valid Usage**

- **VUID-ObjectToWorldKHR-ObjectToWorldKHR-04305**
  The `ObjectToWorldKHR` decoration must be used only within the `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR` Execution Model

- **VUID-ObjectToWorldKHR-ObjectToWorldKHR-04306**
  The variable decorated with `ObjectToWorldKHR` must be declared using the `Input Storage Class`

- **VUID-ObjectToWorldKHR-ObjectToWorldKHR-04307**
  The variable decorated with `ObjectToWorldKHR` must be declared as a matrix with four columns of three-component vectors of 32-bit floating-point values

**PatchVertices**

Decorating a variable with the `PatchVertices` built-in decoration will make that variable contain the number of vertices in the input patch being processed by the shader. In a Tessellation Control Shader, this is the same as the `name:patchControlPoints` member of `VkPipelineTessellationStateCreateInfo`. In a Tessellation Evaluation Shader, `PatchVertices` is equal to the tessellation control output patch size. When the same shader is used in different pipelines where the patch sizes are configured differently, the value of the `PatchVertices` variable will also differ.

**Valid Usage**

- **VUID-PatchVertices-PatchVertices-04308**
  The `PatchVertices` decoration must be used only within the `TessellationControl` or `TessellationEvaluation Execution Model`

- **VUID-PatchVertices-PatchVertices-04309**
  The variable decorated with `PatchVertices` must be declared using the `Input Storage Class`

- **VUID-PatchVertices-PatchVertices-04310**
  The variable decorated with `PatchVertices` must be declared as a scalar 32-bit integer value

**PointCoord**

Decorating a variable with the `PointCoord` built-in decoration will make that variable contain the coordinate of the current fragment within the point being rasterized, normalized to the size of the point with origin in the upper left corner of the point, as described in Basic Point Rasterization. If the primitive the fragment shader invocation belongs to is not a point, then the variable decorated with `PointCoord` contains an undefined value.

**Note**

Depending on how the point is rasterized, `PointCoord` may never reach (0,0) or (1,1).
Valid Usage

- VUID-PointCoord-PointCoord-04311
  The **PointCoord** decoration **must** be used only within the **Fragment Execution Model**

- VUID-PointCoord-PointCoord-04312
  The variable decorated with **PointCoord** **must** be declared using the **Input Storage Class**

- VUID-PointCoord-PointCoord-04313
  The variable decorated with **PointCoord** **must** be declared as a two-component vector of 32-bit floating-point values

**PointSize**

Decorating a variable with the **PointSize** built-in decoration will make that variable contain the size of point primitives. The value written to the variable decorated with **PointSize** by the last vertex processing stage in the pipeline is used as the framebuffer-space size of points produced by rasterization.

*Note*

When **PointSize** decorates a variable in the **Input Storage Class**, it contains the data written to the output variable decorated with **PointSize** from the previous shader stage.

Valid Usage

- VUID-PointSize-PointSize-04314
  The **PointSize** decoration **must** be used only within the **MeshNV, Vertex, TessellationControl, TessellationEvaluation, or Geometry Execution Model**

- VUID-PointSize-PointSize-04315
  The variable decorated with **PointSize** within the **MeshNV or Vertex Execution Model** **must** be declared using the **Output Storage Class**

- VUID-PointSize-PointSize-04316
  The variable decorated with **PointSize** within the **TessellationControl, TessellationEvaluation, or Geometry Execution Model** **must** not be declared using a **Storage Class** other than **Input or Output**

- VUID-PointSize-PointSize-04317
  The variable decorated with **PointSize** **must** be declared as a scalar 32-bit floating-point value

**Position**

Decorating a variable with the **Position** built-in decoration will make that variable contain the position of the current vertex. In the last vertex processing stage, the value of the variable decorated with **Position** is used in subsequent primitive assembly, clipping, and rasterization operations.
Note
When `Position` decorates a variable in the `Input Storage Class`, it contains the data written to the output variable decorated with `Position` from the previous shader stage.

Valid Usage

- **VUID-Position-Position-04318**
  The `Position` decoration **must** be used only within the `MeshNV, Vertex, TessellationControl, TessellationEvaluation, or Geometry Execution Model`

- **VUID-Position-Position-04319**
  The variable decorated with `Position` within `MeshNV or Vertex Execution Model` **must** be declared using the `Output Storage Class`

- **VUID-Position-Position-04320**
  The variable decorated with `Position` within `TessellationControl, TessellationEvaluation, or Geometry Execution Model` **must** not be declared using a `Storage Class` other than `Input` or `Output`

- **VUID-Position-Position-04321**
  The variable decorated with `Position` **must** be declared as a four-component vector of 32-bit floating-point values

`PositionPerViewNV`

Decorating a variable with the `PositionPerViewNV` built-in decoration will make that variable contain the position of the current vertex, for each view.

Elements of the array correspond to views in a multiview subpass, and those elements corresponding to views in the view mask of the subpass the shader is compiled against will be used as the position value for those views. For the final vertex processing stage in the pipeline, values written to an output variable decorated with `PositionPerViewNV` are used in subsequent primitive assembly, clipping, and rasterization operations, as with `Position`. `PositionPerViewNV` output in an earlier vertex processing stage is available as an input in the subsequent vertex processing stage.

If a shader is compiled against a subpass that has the `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX` bit set, then the position values for each view **must** not differ in any component other than the X component. If the values do differ, one will be chosen in an implementation-dependent manner.
Valid Usage

- VUID-PositionPerViewNV-PositionPerViewNV-04322
  The `PositionPerViewNV` decoration must be used only within the `MeshNV`, `Vertex`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`.

- VUID-PositionPerViewNV-PositionPerViewNV-04323
  The variable decorated with `PositionPerViewNV` within the `Vertex`, or `MeshNV Execution Model` must be declared using the `Output Storage Class`.

- VUID-PositionPerViewNV-PositionPerViewNV-04324
  The variable decorated with `PositionPerViewNV` within the `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model` must not be declared using a `Storage Class` other than `Input` or `Output`.

- VUID-PositionPerViewNV-PositionPerViewNV-04325
  The variable decorated with `PositionPerViewNV` must be declared as an array of four-component vector of 32-bit floating-point values with at least as many elements as the maximum view in the subpass's view mask plus one.

- VUID-PositionPerViewNV-PositionPerViewNV-04326
  The array variable decorated with `PositionPerViewNV` must only be indexed by a constant or specialization constant.

**PrimitiveCountNV**

Decorating a variable with the `PrimitiveCountNV` decoration will make that variable contain the primitive count. The primitive count specifies the number of primitives in the output mesh produced by the mesh shader that will be processed by subsequent pipeline stages.

Valid Usage

- VUID-PrimitiveCountNV-PrimitiveCountNV-04327
  The `PrimitiveCountNV` decoration must be used only within the `MeshNV Execution Model`.

- VUID-PrimitiveCountNV-PrimitiveCountNV-04328
  The variable decorated with `PrimitiveCountNV` must be declared using the `Output Storage Class`.

- VUID-PrimitiveCountNV-PrimitiveCountNV-04329
  The variable decorated with `PrimitiveCountNV` must be declared as a scalar 32-bit integer value.

**PrimitiveId**

Decorating a variable with the `PrimitiveId` built-in decoration will make that variable contain the index of the current primitive.

The index of the first primitive generated by a drawing command is zero, and the index is incremented after every individual point, line, or triangle primitive is processed.

For triangles drawn as points or line segments (see `Polygon Mode`), the primitive index is
incremented only once, even if multiple points or lines are eventually drawn.

Variables decorated with \texttt{PrimitiveId} are reset to zero between each instance drawn.

Restarting a primitive topology using primitive restart has no effect on the value of variables decorated with \texttt{PrimitiveId}.

In tessellation control and tessellation evaluation shaders, it will contain the index of the patch within the current set of rendering primitives that corresponds to the shader invocation.

In a geometry shader, it will contain the number of primitives presented as input to the shader since the current set of rendering primitives was started.

In a fragment shader, it will contain the primitive index written by the geometry shader if a geometry shader is present, or with the value that would have been presented as input to the geometry shader had it been present.

In an intersection, any-hit, or closest hit shader, it will contain the index within the geometry of the triangle or bounding box being processed.

\begin{quote}
\textbf{Note}

When the \texttt{PrimitiveId} decoration is applied to an output variable in the mesh shader or geometry shader, the resulting value is seen through the \texttt{PrimitiveId} decorated input variable in the fragment shader.

The fragment shader using \texttt{PrimitiveId} will need to declare either the \texttt{Geometry} or \texttt{Tessellation} capability to satisfy the requirement SPIR-V has to use \texttt{PrimitiveId}.
\end{quote}
Valid Usage

- **VUID-PrimitiveId-PrimitiveId-04330**
  The `PrimitiveId` decoration must be used only within the `MeshNV`, `IntersectionKHR`, `AnyHitKHR`, `ClosestHitKHR`, `TessellationControl`, `TessellationEvaluation`, `Geometry`, or `Fragment Execution Model`.

- **VUID-PrimitiveId-Fragment-04331**
  If pipeline contains both the `Fragment` and `Geometry Execution Model` and a variable decorated with `PrimitiveId` is read from `Fragment` shader, then the `Geometry` shader must write to the output variables decorated with `PrimitiveId` in all execution paths.

- **VUID-PrimitiveId-Fragment-04332**
  If pipeline contains both the `Fragment` and `MeshNV Execution Model` and a variable decorated with `PrimitiveId` is read from `Fragment` shader, then the `MeshNV` shader must write to the output variables decorated with `PrimitiveId` in all execution paths.

- **VUID-PrimitiveId-Fragment-04333**
  If `Fragment Execution Model` contains a variable decorated with `PrimitiveId` either the `Geometry` or `Tessellation` capability must also be declared.

- **VUID-PrimitiveId-PrimitiveId-04334**
  The variable decorated with `PrimitiveId` within the `TessellationControl`, `TessellationEvaluation`, `Fragment`, `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR Execution Model` must be declared using the `Input Storage Class`.

- **VUID-PrimitiveId-PrimitiveId-04335**
  The variable decorated with `PrimitiveId` within the `Geometry Execution Model` must be declared using the `Input` or `Output Storage Class`.

- **VUID-PrimitiveId-PrimitiveId-04336**
  The variable decorated with `PrimitiveId` within the `MeshNV Execution Model` must be declared using the `Output Storage Class`.

- **VUID-PrimitiveId-PrimitiveId-04337**
  The variable decorated with `PrimitiveId` must be declared as a scalar 32-bit integer value.

**PrimitiveIndicesNV**

Decorating a variable with the `PrimitiveIndicesNV` decoration will make that variable contain the output array of vertex index values. Depending on the output primitive type declared using the execution mode, the indices are split into groups of one (OutputPoints), two (OutputLinesNV), or three (OutputTriangles) indices and each group generates a primitive.
**Valid Usage**

- **VUID-PrimitiveIndicesNV-PrimitiveIndicesNV-04338**
  The `PrimitiveIndicesNV` decoration **must** be used only within the `MeshNV Execution Model`.

- **VUID-PrimitiveIndicesNV-PrimitiveIndicesNV-04339**
  The variable decorated with `PrimitiveIndicesNV` **must** be declared using the `Output Storage Class`.

- **VUID-PrimitiveIndicesNV-PrimitiveIndicesNV-04340**
  The variable decorated with `PrimitiveIndicesNV` **must** be declared as an array of scalar 32-bit integer values.

- **VUID-PrimitiveIndicesNV-PrimitiveIndicesNV-04341**
  All index values of the array decorated with `PrimitiveIndicesNV` **must** be in the range \([0, N-1]\), where \(N\) is the value specified by the `OutputVertices Execution Mode`.

- **VUID-PrimitiveIndicesNV-OutputPoints-04342**
  If the **Execution Mode** is `OutputPoints`, then the array decorated with `PrimitiveIndicesNV` must be the size of the value specified by `OutputPrimitivesNV`.

- **VUID-PrimitiveIndicesNV-OutputLinesNV-04343**
  If the **Execution Mode** is `OutputLinesNV`, then the array decorated with `PrimitiveIndicesNV` must be the size of two times the value specified by `OutputPrimitivesNV`.

- **VUID-PrimitiveIndicesNV-OutputTrianglesNV-04344**
  If the **Execution Mode** is `OutputTrianglesNV`, then the array decorated with `PrimitiveIndicesNV` must be the size of three times the value specified by `OutputPrimitivesNV`.

**PrimitiveShadingRateKHR**

Decorating a variable with the `PrimitiveShadingRateKHR` built-in decoration will make that variable contain the **primitive fragment shading rate**.

The value written to the variable decorated with `PrimitiveShadingRateKHR` by the last vertex processing stage in the pipeline is used as the **primitive fragment shading rate**. Outputs in previous shader stages are ignored.

If the last active vertex processing stage shader entry point's interface does not include a variable decorated with `PrimitiveShadingRateKHR`, then it is as if the shader specified a fragment shading rate value of 0, indicating a horizontal and vertical rate of 1 pixel.

If a shader has `PrimitiveShadingRateKHR` in the output interface and there is an execution path through the shader that does not write to it, its value is undefined for executions of the shader that take that path.
Valid Usage

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04484
  The `PrimitiveShadingRateKHR` decoration must be used only within the MeshNV, Vertex, or Geometry Execution Model.

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04485
  The variable decorated with `PrimitiveShadingRateKHR` must be declared using the Output Storage Class.

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04486
  The variable decorated with `PrimitiveShadingRateKHR` must be declared as a scalar 32-bit integer value.

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04487
  The value written to `PrimitiveShadingRateKHR` must include no more than one of `Vertical2Pixels` and `Vertical4Pixels`.

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04488
  The value written to `PrimitiveShadingRateKHR` must include no more than one of `Horizontal2Pixels` and `Horizontal4Pixels`.

- VUID-PrimitiveShadingRateKHR-PrimitiveShadingRateKHR-04489
  The value written to `PrimitiveShadingRateKHR` must not have any bits set other than those defined by Fragment Shading Rate Flags enumerants in the SPIR-V specification.

RayGeometryIndexKHR

A variable decorated with the `RayGeometryIndexKHR` decoration will contain the geometry index for the acceleration structure geometry currently being shaded.

Valid Usage

- VUID-RayGeometryIndexKHR-RayGeometryIndexKHR-04345
  The `RayGeometryIndexKHR` decoration must be used only within the IntersectionKHR, AnyHitKHR, or ClosestHitKHR Execution Model.

- VUID-RayGeometryIndexKHR-RayGeometryIndexKHR-04346
  The variable decorated with `RayGeometryIndexKHR` must be declared using the Input Storage Class.

- VUID-RayGeometryIndexKHR-RayGeometryIndexKHR-04347
  The variable decorated with `RayGeometryIndexKHR` must be declared as a scalar 32-bit integer value.

RayTmaxKHR

A variable decorated with the `RayTmaxKHR` decoration will contain the parametric $t_{\text{max}}$ values of the ray being processed. The values are independent of the space in which the ray origin and direction exist.

The $t_{\text{max}}$ value changes throughout the lifetime of the ray query that produced the intersection.
In the closest hit shader, the value reflects the closest distance to the intersected primitive. In the any-hit shader, it reflects the distance to the primitive currently being intersected. In the intersection shader, it reflects the distance to the closest primitive intersected so far. The value can change in the intersection shader after calling `OpReportIntersectionKHR` if the corresponding any-hit shader does not ignore the intersection. In a miss shader, the value is identical to the parameter passed into `OpTraceRayKHR`.

### Valid Usage

- **VUID-RayTmaxKHR-RayTmaxKHR-04348**
  The `RayTmaxKHR` decoration **must** be used only within the `IntersectionKHR`, `AnyHitKHR`, `ClosestHitKHR`, or `MissKHR` Execution Model

- **VUID-RayTmaxKHR-RayTmaxKHR-04349**
  The variable decorated with `RayTmaxKHR` **must** be declared using the `Input Storage Class`

- **VUID-RayTmaxKHR-RayTmaxKHR-04350**
  The variable decorated with `RayTmaxKHR` **must** be declared as a scalar 32-bit floating-point value

### RayTminKHR

A variable decorated with the `RayTminKHR` decoration will contain the parametric $t_{\text{min}}$ values of the ray being processed. The values are independent of the space in which the ray origin and direction exist.

The $t_{\text{min}}$ value remains constant for the duration of the ray query.

### Valid Usage

- **VUID-RayTminKHR-RayTminKHR-04351**
  The `RayTminKHR` decoration **must** be used only within the `IntersectionKHR`, `AnyHitKHR`, `ClosestHitKHR`, or `MissKHR` Execution Model

- **VUID-RayTminKHR-RayTminKHR-04352**
  The variable decorated with `RayTminKHR` **must** be declared using the `Input Storage Class`

- **VUID-RayTminKHR-RayTminKHR-04353**
  The variable decorated with `RayTminKHR` **must** be declared as a scalar 32-bit floating-point value

### SampleId

Decorating a variable with the `SampleId` built-in decoration will make that variable contain the **coverage index** for the current fragment shader invocation. `SampleId` ranges from zero to the number of samples in the framebuffer minus one. If a fragment shader entry point's interface includes an input variable decorated with `SampleId`, **Sample Shading** is considered enabled with a `minSampleShading` value of 1.0.
Valid Usage

- VUID-SampleId-SampleId-04354
  The **SampleId** decoration must be used only within the Fragment Execution Model

- VUID-SampleId-SampleId-04355
  The variable decorated with **SampleId** must be declared using the Input Storage Class

- VUID-SampleId-SampleId-04356
  The variable decorated with **SampleId** must be declared as a scalar 32-bit integer value

**SampleMask**

Decorating a variable with the **SampleMask** built-in decoration will make any variable contain the coverage mask for the current fragment shader invocation.

A variable in the Input storage class decorated with **SampleMask** will contain a bitmask of the set of samples covered by the primitive generating the fragment during rasterization. It has a sample bit set if and only if the sample is considered covered for this fragment shader invocation. **SampleMask[]** is an array of integers. Bits are mapped to samples in a manner where bit B of mask M (**SampleMask[M]**) corresponds to sample $32 \times M + B$.

When state specifies multiple fragment shader invocations for a given fragment, the sample mask for any single fragment shader invocation specifies the subset of the covered samples for the fragment that correspond to the invocation. In this case, the bit corresponding to each covered sample will be set in exactly one fragment shader invocation.

If the PostDepthCoverage execution mode is specified, the sample is considered covered if and only if the sample is covered by the primitive, and the sample is still covered after depth testing. Otherwise the sample is considered covered if the sample is covered by the primitive, regardless of the result of the fragment tests.

A variable in the Output storage class decorated with **SampleMask** is an array of integers forming a bit array in a manner similar to an input variable decorated with **SampleMask**, but where each bit represents coverage as computed by the shader. Modifying the sample mask by writing zero to a bit of **SampleMask** causes the sample to be considered uncovered. If this variable is also decorated with OverrideCoverageNV, the fragment coverage is replaced with the sample mask bits set in the shader otherwise the fragment coverage is ANDed with the bits of the sample mask. If the fragment shader is being evaluated at any frequency other than per-fragment, bits of the sample mask not corresponding to the current fragment shader invocation are ignored. This array must be sized in the fragment shader either implicitly or explicitly, to be no larger than the implementation-dependent maximum sample-mask (as an array of 32-bit elements), determined by the maximum number of samples. If a fragment shader entry point's interface includes an output variable decorated with **SampleMask**, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a fragment shader entry point's interface does not include an output variable decorated with **SampleMask**, the sample mask has no effect on the processing of a fragment.
**Valid Usage**

- **VUID-SampleMask-SampleMask-04357**
  The **SampleMask** decoration **must** be used only within the **Fragment Execution Model**

- **VUID-SampleMask-SampleMask-04358**
  The variable decorated with **SampleMask** **must** be declared using the **Input** or **Output Storage Class**

- **VUID-SampleMask-SampleMask-04359**
  The variable decorated with **SampleMask** **must** be declared as an array of 32-bit integer values

**SamplePosition**

Decorating a variable with the **SamplePosition** built-in decoration will make that variable contain the sub-pixel position of the sample being shaded. The top left of the pixel is considered to be at coordinate (0,0) and the bottom right of the pixel is considered to be at coordinate (1,1).

If the render pass has a fragment density map attachment, the variable will instead contain the sub-fragment position of the sample being shaded. The top left of the fragment is considered to be at coordinate (0,0) and the bottom right of the fragment is considered to be at coordinate (1,1) for any fragment area.

If a fragment shader entry point's interface includes an input variable decorated with **SamplePosition**, **Sample Shading** is considered enabled with a **minSampleShading** value of 1.0.

If the current pipeline uses **custom sample locations** the value of any variable decorated with the **SamplePosition** built-in decoration is undefined.

**Valid Usage**

- **VUID-SamplePosition-SamplePosition-04360**
  The **SamplePosition** decoration **must** be used only within the **Fragment Execution Model**

- **VUID-SamplePosition-SamplePosition-04361**
  The variable decorated with **SamplePosition** **must** be declared using the **Input Storage Class**

- **VUID-SamplePosition-SamplePosition-04362**
  The variable decorated with **SamplePosition** **must** be declared as a two-component vector of 32-bit floating-point values

**ShadingRateKHR**

Decorating a variable with the **ShadingRateKHR** built-in decoration will make that variable contain the **fragment shading rate** for the current fragment invocation.
**Valid Usage**

- VUID-ShadingRateKHR-ShadingRateKHR-04490
  The *ShadingRateKHR* decoration **must** be used only within the *Fragment Execution Model*

- VUID-ShadingRateKHR-ShadingRateKHR-04491
  The variable decorated with *ShadingRateKHR* **must** be declared using the *Input Storage Class*

- VUID-ShadingRateKHR-ShadingRateKHR-04492
  The variable decorated with *ShadingRateKHR* **must** be declared as a scalar 32-bit integer value

**SMCountNV**

Decorating a variable with the *SMCountNV* built-in decoration will make that variable contain the number of SMs on the device.

**Valid Usage**

- VUID-SMCountNV-SMCountNV-04363
  The variable decorated with *SMCountNV* **must** be declared using the *Input Storage Class*

- VUID-SMCountNV-SMCountNV-04364
  The variable decorated with *SMCountNV* **must** be declared as a scalar 32-bit integer value

**SMIDNV**

Decorating a variable with the *SMIDNV* built-in decoration will make that variable contain the ID of the SM on which the current shader invocation is running. This variable is in the range [0, *SMCountNV*-1].

**Valid Usage**

- VUID-SMIDNV-SMIDNV-04365
  The variable decorated with *SMIDNV* **must** be declared using the *Input Storage Class*

- VUID-SMIDNV-SMIDNV-04366
  The variable decorated with *SMIDNV* **must** be declared as a scalar 32-bit integer value

**SubgroupId**

Decorating a variable with the *SubgroupId* built-in decoration will make that variable contain the index of the subgroup within the local workgroup. This variable is in range [0, *NumSubgroups*-1].
Valid Usage

- VUID-SubgroupId-SubgroupId-04367
  The `SubgroupId` decoration must be used only within the `GLCompute`, `MeshNV`, or `TaskNV Execution Model`.

- VUID-SubgroupId-SubgroupId-04368
  The variable decorated with `SubgroupId` must be declared using the `Input Storage Class`.

- VUID-SubgroupId-SubgroupId-04369
  The variable decorated with `SubgroupId` must be declared as a scalar 32-bit integer value.

SubgroupEqMask

Decorating a variable with the `SubgroupEqMask` builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bit corresponding to the `SubgroupLocalInvocationId` is set in the variable decorated with `SubgroupEqMask`. All other bits are set to zero.

`SubgroupEqMaskKHR` is an alias of `SubgroupEqMask`.

Valid Usage

- VUID-SubgroupEqMask-SubgroupEqMask-04370
  The variable decorated with `SubgroupEqMask` must be declared using the `Input Storage Class`.

- VUID-SubgroupEqMask-SubgroupEqMask-04371
  The variable decorated with `SubgroupEqMask` must be declared as a four-component vector of 32-bit integer values.

SubgroupGeMask

Decorating a variable with the `SubgroupGeMask` builtin decoration will make that variable contain the subgroup mask of the current subgroup invocation. The bits corresponding to the invocations greater than or equal to `SubgroupLocalInvocationId` through `SubgroupSize-1` are set in the variable decorated with `SubgroupGeMask`. All other bits are set to zero.

`SubgroupGeMaskKHR` is an alias of `SubgroupGeMask`.

Valid Usage

- VUID-SubgroupGeMask-SubgroupGeMask-04372
  The variable decorated with `SubgroupGeMask` must be declared using the `Input Storage Class`.

- VUID-SubgroupGeMask-SubgroupGeMask-04373
  The variable decorated with `SubgroupGeMask` must be declared as a four-component vector of 32-bit integer values.
SubgroupGtMask

Decorating a variable with the SubgroupGtMask builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations greater than SubgroupLocalInvocationId through SubgroupSize-1 are set in the variable decorated with SubgroupGtMask. All other bits are set to zero.

SubgroupGtMaskKHR is an alias of SubgroupGtMask.

Valid Usage

- VUID-SubgroupGtMask-SubgroupGtMask-04374
  The variable decorated with SubgroupGtMask must be declared using the Input Storage Class
- VUID-SubgroupGtMask-SubgroupGtMask-04375
  The variable decorated with SubgroupGtMask must be declared as a four-component vector of 32-bit integer values

SubgroupLeMask

Decorating a variable with the SubgroupLeMask builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations less than or equal to SubgroupLocalInvocationId are set in the variable decorated with SubgroupLeMask. All other bits are set to zero.

SubgroupLeMaskKHR is an alias of SubgroupLeMask.

Valid Usage

- VUID-SubgroupLeMask-SubgroupLeMask-04376
  The variable decorated with SubgroupLeMask must be declared using the Input Storage Class
- VUID-SubgroupLeMask-SubgroupLeMask-04377
  The variable decorated with SubgroupLeMask must be declared as a four-component vector of 32-bit integer values

SubgroupLtMask

Decorating a variable with the SubgroupLtMask builtin decoration will make that variable contain the *subgroup mask* of the current subgroup invocation. The bits corresponding to the invocations less than SubgroupLocalInvocationId are set in the variable decorated with SubgroupLtMask. All other bits are set to zero.

SubgroupLtMaskKHR is an alias of SubgroupLtMask.
Valid Usage

- VUID-SubgroupLtMask-SubgroupLtMask-04378
  The variable decorated with **SubgroupLtMask** must be declared using the **Input Storage Class**

- VUID-SubgroupLtMask-SubgroupLtMask-04379
  The variable decorated with **SubgroupLtMask** must be declared as a four-component vector of 32-bit integer values

**SubgroupLocalInvocationId**

Decorating a variable with the **SubgroupLocalInvocationId** builtin decoration will make that variable contain the index of the invocation within the subgroup. This variable is in range \([0, \text{SubgroupSize}-1]\).

*Note*

There is no direct relationship between **SubgroupLocalInvocationId** and **LocalInvocationId** or **LocalInvocationIndex**. If the pipeline was created with \text{VK\_PIPELINE\_SHADER\_STAGE\_CREATE\_REQUIRE\_FULL\_SUBGROUPS\_BIT\_EXT}, applications can compute their own local invocation index to serve the same purpose:

\[
\text{index} = \text{SubgroupLocalInvocationId} + \text{SubgroupId} \times \text{SubgroupSize}
\]

If full subgroups are not enabled, some subgroups may be dispatched with inactive invocations that don't correspond to a local workgroup invocation, making the value of index unreliable.

Valid Usage

- VUID-SubgroupLocalInvocationId-SubgroupLocalInvocationId-04380
  The variable decorated with **SubgroupLocalInvocationId** must be declared using the **Input Storage Class**

- VUID-SubgroupLocalInvocationId-SubgroupLocalInvocationId-04381
  The variable decorated with **SubgroupLocalInvocationId** must be declared as a scalar 32-bit integer value

**SubgroupSize**

Decorating a variable with the **SubgroupSize** builtin decoration will make that variable contain the implementation-dependent number of invocations in a subgroup. This value must be a power-of-two integer.

If the pipeline was created with \text{VK\_PIPELINE\_SHADER\_STAGE\_CREATE\_ALLOW\_VARYING\_SUBGROUP\_SIZE\_BIT\_EXT} flag set, the **SubgroupSize** decorated variable will contain the subgroup size for each subgroup that gets dispatched. This value must be between \text{minSubgroupSize} and \text{maxSubgroupSize} and must be uniform with subgroup scope. The value may vary across a single draw or dispatch call, and for fragment
shaders may vary across a single primitive.

If the pipeline was created with a chained `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure, the `SubgroupSize` decorated variable will match `requiredSubgroupSize`.

If the pipeline was not created with the `VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT` flag set and no `VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT` structure was chained, the variable decorated with `SubgroupSize` will match `subgroupSize`.

The maximum number of invocations that an implementation can support per subgroup is 128.

**Valid Usage**

- **VUID-SubgroupSize-SubgroupSize-04382**
  The variable decorated with `SubgroupSize` must be declared using the Input Storage Class

- **VUID-SubgroupSize-SubgroupSize-04383**
  The variable decorated with `SubgroupSize` must be declared as a scalar 32-bit integer value

**TaskCountNV**

Decorating a variable with the `TaskCountNV` decoration will make that variable contain the task count. The task count specifies the number of subsequent mesh shader workgroups that get generated upon completion of the task shader.

**Valid Usage**

- **VUID-TaskCountNV-TaskCountNV-04384**
  The `TaskCountNV` decoration must be used only within the TaskNV Execution Model

- **VUID-TaskCountNV-TaskCountNV-04385**
  The variable decorated with `TaskCountNV` must be declared using the Output Storage Class

- **VUID-TaskCountNV-TaskCountNV-04386**
  The variable decorated with `TaskCountNV` must be declared as a scalar 32-bit integer value

**TessCoord**

Decorating a variable with the `TessCoord` built-in decoration will make that variable contain the three-dimensional (u,v,w) barycentric coordinate of the tessellated vertex within the patch. u, v, and w are in the range [0,1] and vary linearly across the primitive being subdivided. For the tessellation modes of `Quads` or `IsoLines`, the third component is always zero.
Valid Usage

- VUID-TessCoord-TessCoord-04387
  The TessCoord decoration must be used only within the TessellationEvaluation Execution Model

- VUID-TessCoord-TessCoord-04388
  The variable decorated with TessCoord must be declared using the Input Storage Class

- VUID-TessCoord-TessCoord-04389
  The variable decorated with TessCoord must be declared as a three-component vector of 32-bit floating-point values

TessLevelOuter

Decorating a variable with the TessLevelOuter built-in decoration will make that variable contain the outer tessellation levels for the current patch.

In tessellation control shaders, the variable decorated with TessLevelOuter can be written to, which controls the tessellation factors for the resulting patch. These values are used by the tessellator to control primitive tessellation and can be read by tessellation evaluation shaders.

In tessellation evaluation shaders, the variable decorated with TessLevelOuter can read the values written by the tessellation control shader.

Valid Usage

- VUID-TessLevelOuter-TessLevelOuter-04390
  The TessLevelOuter decoration must be used only within the TessellationControl or TessellationEvaluation Execution Model

- VUID-TessLevelOuter-TessLevelOuter-04391
  The variable decorated with TessLevelOuter within the TessellationControl Execution Model must be declared using the Output Storage Class

- VUID-TessLevelOuter-TessLevelOuter-04392
  The variable decorated with TessLevelOuter within the TessellationEvaluation Execution Model must be declared using the Input Storage Class

- VUID-TessLevelOuter-TessLevelOuter-04393
  The variable decorated with TessLevelOuter must be declared as an array of size four, containing 32-bit floating-point values

TessLevelInner

Decorating a variable with the TessLevelInner built-in decoration will make that variable contain the inner tessellation levels for the current patch.

In tessellation control shaders, the variable decorated with TessLevelInner can be written to, which controls the tessellation factors for the resulting patch. These values are used by the tessellator to control primitive tessellation and can be read by tessellation evaluation shaders.
In tessellation evaluation shaders, the variable decorated with `TessLevelInner` can read the values written by the tessellation control shader.

**Valid Usage**

- **VUID-TessLevelInner-TessLevelInner-04394**
  The `TessLevelInner` decoration must be used only within the `TessellationControl` or `TessellationEvaluation Execution Model`

- **VUID-TessLevelInner-TessLevelInner-04395**
  The variable decorated with `TessLevelInner` within the `TessellationControl Execution Model` must be declared using the `Output Storage Class`

- **VUID-TessLevelInner-TessLevelInner-04396**
  The variable decorated with `TessLevelInner` within the `TessellationEvaluation Execution Model` must be declared using the `Input Storage Class`

- **VUID-TessLevelInner-TessLevelInner-04397**
  The variable decorated with `TessLevelInner` must be declared as an array of size two, containing 32-bit floating-point values

---

**VertexIndex**

Decorating a variable with the `VertexIndex` built-in decoration will make that variable contain the index of the vertex that is being processed by the current vertex shader invocation. For non-indexed draws, this variable begins at the `firstVertex` parameter to `vkCmdDraw` or the `firstVertex` member of a structure consumed by `vkCmdDrawIndirect` and increments by one for each vertex in the draw. For indexed draws, its value is the content of the index buffer for the vertex plus the `vertexOffset` parameter to `vkCmdDrawIndexed` or the `vertexOffset` member of the structure consumed by `vkCmdDrawIndexedIndirect`.

**Note**

`VertexIndex` starts at the same starting value for each instance.

**Valid Usage**

- **VUID-VertexIndex-VertexIndex-04398**
  The `VertexIndex` decoration must be used only within the `Vertex Execution Model`

- **VUID-VertexIndex-VertexIndex-04399**
  The variable decorated with `VertexIndex` must be declared using the `Input Storage Class`

- **VUID-VertexIndex-VertexIndex-04400**
  The variable decorated with `VertexIndex` must be declared as a scalar 32-bit integer value

---

**ViewIndex**

The `ViewIndex` decoration can be applied to a shader input which will be filled with the index of the view that is being processed by the current shader invocation.
If multiview is enabled in the render pass, this value will be one of the bits set in the view mask of the subpass the pipeline is compiled against. If multiview is not enabled in the render pass, this value will be zero.

**Valid Usage**

- **VUID-ViewIndex-ViewIndex-04401**
  The ViewIndex decoration must not be used within the GLCompute Execution Model

- **VUID-ViewIndex-ViewIndex-04402**
  The variable decorated with ViewIndex must be declared using the Input Storage Class

- **VUID-ViewIndex-ViewIndex-04403**
  The variable decorated with ViewIndex must be declared as a scalar 32-bit integer value

**ViewportIndex**

Decorating a variable with the ViewportIndex built-in decoration will make that variable contain the index of the viewport.

In a mesh, vertex, tessellation evaluation, or geometry shader, the variable decorated with ViewportIndex can be written to with the viewport index to which the primitive produced by that shader will be directed.

The selected viewport index is used to select the viewport transform, scissor rectangle, and exclusive scissor rectangle.

The last active vertex processing stage (in pipeline order) controls the ViewportIndex that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the ViewportIndex.

If the last active vertex processing stage shader entry point’s interface does not include a variable decorated with ViewportIndex, then the first viewport is used. If a vertex processing stage shader entry point’s interface includes a variable decorated with ViewportIndex, it must write the same value to ViewportIndex for all output vertices of a given primitive.

In a fragment shader, the variable decorated with ViewportIndex contains the viewport index of the primitive that the fragment invocation belongs to.
Valid Usage

- VUID-ViewportIndex-ViewportIndex-04404
  The `ViewportIndex` decoration must be used only within the `MeshNV`, `Vertex`, `TessellationEvaluation`, `Geometry`, or `Fragment Execution Model`.

- VUID-ViewportIndex-ViewportIndex-04406
  The variable decorated with `ViewportIndex` within the `MeshNV`, `Vertex`, `TessellationEvaluation`, or `Geometry Execution Model` must be declared using the `Output Storage Class`.

- VUID-ViewportIndex-ViewportIndex-04407
  The variable decorated with `ViewportIndex` within the `Fragment Execution Model` must be declared using the `Input Storage Class`.

- VUID-ViewportIndex-ViewportIndex-04408
  The variable decorated with `ViewportIndex` must be declared as a scalar 32-bit integer value.

ViewportMaskNV

Decorating a variable with the `ViewportMaskNV` built-in decoration will make that variable contain the viewport mask.

In a mesh, vertex, tessellation evaluation, or geometry shader, the variable decorated with `ViewportMaskNV` can be written to with the mask of which viewports the primitive produced by that shader will directed.

The `ViewportMaskNV` variable must be an array that has `⌈(VkPhysicalDeviceLimits::maxViewports / 32)⌉` elements. When a shader writes to this variable, bit B of element M controls whether a primitive is emitted to viewport `32 × M + B`. The viewports indicated by the mask are used to select the viewport transform, scissor rectangle, and exclusive scissor rectangle that a primitive will be transformed by.

The last active `vertex processing stage` (in pipeline order) controls the `ViewportMaskNV` that is used. Outputs in previous shader stages are not used, even if the last stage fails to write the `ViewportMaskNV`. When `ViewportMaskNV` is written by the final vertex processing stage, any variable decorated with `ViewportIndex` in the fragment shader will have the index of the viewport that was used in generating that fragment.

If a vertex processing stage shader entry point's interface includes a variable decorated with `ViewportMaskNV`, it must write the same value to `ViewportMaskNV` for all output vertices of a given primitive.
Valid Usage

- VUID-ViewportMaskNV-ViewportMaskNV-04409
  The `ViewportMaskNV` decoration **must** be used only within the `Vertex`, `MeshNV`, `TessellationEvaluation`, or `Geometry Execution Model`

- VUID-ViewportMaskNV-ViewportMaskNV-04410
  The variable decorated with `ViewportMaskNV` **must** be declared using the `Output Storage Class`

- VUID-ViewportMaskNV-ViewportMaskNV-04411
  The variable decorated with `ViewportMaskNV` **must** be declared as an array of 32-bit integer values

**ViewportMaskPerViewNV**

Decorating a variable with the `ViewportMaskPerViewNV` built-in decoration will make that variable contain the mask of viewports primitives are broadcast to, for each view.

The value written to an element of `ViewportMaskPerViewNV` in the last vertex processing stage is a bitmask indicating which viewports the primitive will be directed to. The primitive will be broadcast to the viewport corresponding to each non-zero bit of the bitmask, and that viewport index is used to select the viewport transform, scissor rectangle, and exclusive scissor rectangle, for each view. The same values **must** be written to all vertices in a given primitive, or else the set of viewports used for that primitive is undefined.

Elements of the array correspond to views in a multiview subpass, and those elements corresponding to views in the view mask of the subpass the shader is compiled against will be used as the viewport mask value for those views. `ViewportMaskPerViewNV` output in an earlier vertex processing stage is not available as an input in the subsequent vertex processing stage.

Although `ViewportMaskNV` is an array, `ViewportMaskPerViewNV` is not a two-dimensional array. Instead, `ViewportMaskPerViewNV` is limited to 32 viewports.
Valid Usage

- VUID-ViewportMaskPerViewNV-ViewportMaskPerViewNV-04412
  The `ViewportMaskPerViewNV` decoration must be used only within the `Vertex`, `MeshNV`, `TessellationControl`, `TessellationEvaluation`, or `Geometry Execution Model`.

- VUID-ViewportMaskPerViewNV-ViewportMaskPerViewNV-04413
  The variable decorated with `ViewportMaskPerViewNV` must be declared using the `Output Storage Class`.

- VUID-ViewportMaskPerViewNV-ViewportMaskPerViewNV-04414
  The variable decorated with `ViewportMaskPerViewNV` must be declared as an array of 32-bit integer values.

- VUID-ViewportMaskPerViewNV-ViewportMaskPerViewNV-04415
  The array decorated with `ViewportMaskPerViewNV` must be a size less than or equal to 32.

- VUID-ViewportMaskPerViewNV-ViewportMaskPerViewNV-04416
  The array decorated with `ViewportMaskPerViewNV` must be a size greater than the maximum view in the subpass's view mask.

- VUID-ViewportMaskPerViewNV-ViewportMaskPerViewNV-04417
  The array variable decorated with `ViewportMaskPerViewNV` must only be indexed by a constant or specialization constant.

**WarpsPerSMNV**

Decorating a variable with the `WarpsPerSMNV` built-in decoration will make that variable contain the maximum number of warps executing on a SM.

Valid Usage

- VUID-WarpsPerSMNV-WarpsPerSMNV-04418
  The variable decorated with `WarpsPerSMNV` must be declared using the `Input Storage Class`.

- VUID-WarpsPerSMNV-WarpsPerSMNV-04419
  The variable decorated with `WarpsPerSMNV` must be declared as a scalar 32-bit integer value.

**WarpIDNV**

Decorating a variable with the `WarpIDNV` built-in decoration will make that variable contain the ID of the warp on a SM on which the current shader invocation is running. This variable is in the range `[0, WarpsPerSMNV-1]`. 
Valid Usage

- VUID-WarpIDNV-WarpIDNV-04420
  The variable decorated with \texttt{WarpIDNV} must be declared using the \texttt{Input Storage Class}
- VUID-WarpIDNV-WarpIDNV-04421
  The variable decorated with \texttt{WarpIDNV} must be declared as a scalar 32-bit integer value

WorkgroupId

Decorating a variable with the \texttt{WorkgroupId} built-in decoration will make that variable contain the global workgroup that the current invocation is a member of. Each component ranges from a base value to a base + count value, based on the parameters passed into the dispatch commands.

Valid Usage

- VUID-WorkgroupId-WorkgroupId-04422
  The \texttt{WorkgroupId} decoration must be used only within the \texttt{GLCompute}, \texttt{MeshNV}, or \texttt{TaskNV} Execution Model
- VUID-WorkgroupId-WorkgroupId-04423
  The variable decorated with \texttt{WorkgroupId} must be declared using the \texttt{Input Storage Class}
- VUID-WorkgroupId-WorkgroupId-04424
  The variable decorated with \texttt{WorkgroupId} must be declared as a three-component vector of 32-bit integer values

WorkgroupSize

Decorating an object with the \texttt{WorkgroupSize} built-in decoration will make that object contain the dimensions of a local workgroup. If an object is decorated with the \texttt{WorkgroupSize} decoration, this takes precedence over any \texttt{LocalSize} execution mode.

Valid Usage

- VUID-WorkgroupSize-WorkgroupSize-04425
  The \texttt{WorkgroupSize} decoration must be used only within the \texttt{GLCompute}, \texttt{MeshNV}, or \texttt{TaskNV} Execution Model
- VUID-WorkgroupSize-WorkgroupSize-04426
  The variable decorated with \texttt{WorkgroupSize} must be a specialization constant or a constant
- VUID-WorkgroupSize-WorkgroupSize-04427
  The variable decorated with \texttt{WorkgroupSize} must be declared as a three-component vector of 32-bit integer values

WorldRayDirectionKHR

A variable decorated with the \texttt{WorldRayDirectionKHR} decoration will specify the direction of the ray being processed, in world space.
Valid Usage

- **VUID-WorldRayDirectionKHR-WorldRayDirectionKHR-04428**
  The `WorldRayDirectionKHR` decoration **must** be used only within the `IntersectionKHR`, `AnyHitKHR`, `ClosestHitKHR`, or `MissKHR` Execution Model.

- **VUID-WorldRayDirectionKHR-WorldRayDirectionKHR-04429**
  The variable decorated with `WorldRayDirectionKHR` **must** be declared using the Input Storage Class.

- **VUID-WorldRayDirectionKHR-WorldRayDirectionKHR-04430**
  The variable decorated with `WorldRayDirectionKHR` **must** be declared as a three-component vector of 32-bit floating-point values.

**WorldRayOriginKHR**

A variable decorated with the `WorldRayOriginKHR` decoration will specify the origin of the ray being processed, in world space.

Valid Usage

- **VUID-WorldRayOriginKHR-WorldRayOriginKHR-04431**
  The `WorldRayOriginKHR` decoration **must** be used only within the `IntersectionKHR`, `AnyHitKHR`, `ClosestHitKHR`, or `MissKHR` Execution Model.

- **VUID-WorldRayOriginKHR-WorldRayOriginKHR-04432**
  The variable decorated with `WorldRayOriginKHR` **must** be declared using the Input Storage Class.

- **VUID-WorldRayOriginKHR-WorldRayOriginKHR-04433**
  The variable decorated with `WorldRayOriginKHR` **must** be declared as a three-component vector of 32-bit floating-point values.

**WorldToObjectKHR**

A variable decorated with the `WorldToObjectKHR` decoration will contain the current world-to-object transformation matrix, which is determined by the instance of the current intersection.
Valid Usage

- VUID-WorldToObjectKHR-WorldToObjectKHR-04434
  The `WorldToObjectKHR` decoration **must** be used only within the `IntersectionKHR`, `AnyHitKHR`, or `ClosestHitKHR` Execution Model

- VUID-WorldToObjectKHR-WorldToObjectKHR-04435
  The variable decorated with `WorldToObjectKHR` **must** be declared using the **Input Storage Class**

- VUID-WorldToObjectKHR-WorldToObjectKHR-04436
  The variable decorated with `WorldToObjectKHR` **must** be declared as a matrix with four columns of three-component vectors of 32-bit floating-point values
Chapter 16. Image Operations

16.1. Image Operations Overview

Vulkan Image Operations are operations performed by those SPIR-V Image Instructions which take an `OpTypeImage` (representing a `VkImageView`) or `OpTypeSampledImage` (representing a (`VkImageView`, `VkSampler`) pair). Read, write, and atomic operations also take texel coordinates as operands, and return a value based on a neighborhood of texture elements (texels) within the image. Query operations return properties of the bound image or of the lookup itself. The “Depth” operand of `OpTypeImage` is ignored.

**Note**

Texel is a term which is a combination of the words texture and element. Early interactive computer graphics supported texture operations on textures, a small subset of the image operations on images described here. The discrete samples remain essentially equivalent, however, so we retain the historical term texel to refer to them.

Image Operations include the functionality of the following SPIR-V Image Instructions:

- `OpImageSample*` and `OpImageSparseSample*` read one or more neighboring texels of the image, and **filter** the texel values based on the state of the sampler.
  - Instructions with `ImplicitLod` in the name **determine** the LOD used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with `ExplicitLod` in the name **determine** the LOD used in the sampling operation based on additional coordinates.
  - Instructions with `Proj` in the name apply homogeneous **projection** to the coordinates.

- `OpImageFetch` and `OpImageSparseFetch` return a single texel of the image. No sampler is used.

- `OpImage*Gather` and `OpImageSparse*Gather` read neighboring texels and **return a single component** of each.

- `OpImageRead` (and `OpImageSparseRead`) and `OpImageWrite` read and write, respectively, a texel in the image. No sampler is used.

- `OpImageSampleFootprintNV` identifies and returns information about the set of texels in the image that would be accessed by an equivalent `OpImageSample*` instruction.

- Instructions with `Dref` in the name apply **depth comparison** on the texel values.

- Instructions with `Sparse` in the name additionally return a **sparse residency** code.


- `OpImageQueryLod` returns the lod parameters that would be used in a sample operation. The actual operation is not performed.
16.1.1. Texel Coordinate Systems

Images are addressed by texel coordinates. There are three texel coordinate systems:

- normalized texel coordinates \([0.0, 1.0]\)
- unnormalized texel coordinates \([0.0, \text{width} / \text{height} / \text{depth})\)
- integer texel coordinates \([0, \text{width} / \text{height} / \text{depth})\)

SPIR-V `OpImageFetch`, `OpImageSparseFetch`, `OpImageRead`, `OpImageSparseRead`, and `OpImageWrite` instructions use integer texel coordinates. Other image instructions can use either normalized or unnormalized texel coordinates (selected by the `unnormalizedCoordinates` state of the sampler used in the instruction), but there are limitations on what operations, image state, and sampler state is supported. Normalized coordinates are logically converted to unnormalized as part of image operations, and certain steps are only performed on normalized coordinates. The array layer coordinate is always treated as unnormalized even when other coordinates are normalized.

Normalized texel coordinates are referred to as \((s, t, r, q, a)\), with the coordinates having the following meanings:

- \(s\): Coordinate in the first dimension of an image.
- \(t\): Coordinate in the second dimension of an image.
- \(r\): Coordinate in the third dimension of an image.
  - \((s, t, r)\) are interpreted as a direction vector for Cube images.
- \(q\): Fourth coordinate, for homogeneous (projective) coordinates.
- \(a\): Coordinate for array layer.

The coordinates are extracted from the SPIR-V operand based on the dimensionality of the image variable and type of instruction. For Proj instructions, the components are in order \((s, [t, [r, q]])\), with \(t\) and \(r\) being conditionally present based on the Dim of the image. For non- Proj instructions, the coordinates are \((s [,t [,r [,a]]])\), with \(t\) and \(r\) being conditionally present based on the Dim of the image and \(a\) being conditionally present based on the Arrayed property of the image. Projective image instructions are not supported on Arrayed images.

Unnormalized texel coordinates are referred to as \((u, v, w, a)\), with the coordinates having the following meanings:

- \(u\): Coordinate in the first dimension of an image.
- \(v\): Coordinate in the second dimension of an image.
- \(w\): Coordinate in the third dimension of an image.
- \(a\): Coordinate for array layer.

Only the \(u\) and \(v\) coordinates are directly extracted from the SPIR-V operand, because only 1D and 2D (non-Arrayed) dimensionalities support unnormalized coordinates. The components are in order \((u [,v])\), with \(v\) being conditionally present when the dimensionality is 2D. When normalized coordinates are converted to unnormalized coordinates, all four coordinates are used.
Integer texel coordinates are referred to as \((i,j,k,l,n)\), with the coordinates having the following meanings:

- \(i\): Coordinate in the first dimension of an image.
- \(j\): Coordinate in the second dimension of an image.
- \(k\): Coordinate in the third dimension of an image.
- \(l\): Coordinate for array layer.
- \(n\): Index of the sample within the texel.

They are extracted from the SPIR-V operand in order \((i \[,j\] \[,k\] \[,l\] \[,n\])\), with \(j\) and \(k\) conditionally present based on the \texttt{Dim} of the image, and \(l\) conditionally present based on the \texttt{Arrayed} property of the image. \(n\) is conditionally present and is taken from the \texttt{Sample} image operand.

For all coordinate types, unused coordinates are assigned a value of zero.

**Figure 3. Texel Coordinate Systems, Linear Filtering**

The Texel Coordinate Systems - For the example shown of an \(8\times4\) texel two dimensional image.

- Normalized texel coordinates:
  - The \(s\) coordinate goes from 0.0 to 1.0.
  - The \(t\) coordinate goes from 0.0 to 1.0.

- Unnormalized texel coordinates:
  - The \(u\) coordinate within the range 0.0 to 8.0 is within the image, otherwise it is outside the image.
  - The \(v\) coordinate within the range 0.0 to 4.0 is within the image, otherwise it is outside the image.

- Integer texel coordinates:
The i coordinate within the range 0 to 7 addresses texels within the image, otherwise it is outside the image.

The j coordinate within the range 0 to 3 addresses texels within the image, otherwise it is outside the image.

- Also shown for linear filtering:
  - Given the unnormalized coordinates (u,v), the four texels selected are \(i_0j_0, i_1j_0, i_0j_1, \) and \(i_1j_1\).
  - The fractions \(\alpha\) and \(\beta\).
  - Given the offset \(\Delta_i\) and \(\Delta_j\), the four texels selected by the offset are \(i_0j'_0, i_1j'_0, i_0j'_1, \) and \(i_1j'_1\).

\[\text{Note}\]
For formats with reduced-resolution channels, \(\Delta_i\) and \(\Delta_j\) are relative to the resolution of the highest-resolution channel, and therefore may be divided by two relative to the unnormalized coordinate space of the lower-resolution channels.

\[\text{Figure 4. Texel Coordinate Systems, Nearest Filtering}\]

The Texel Coordinate Systems - For the example shown of an 8×4 texel two dimensional image.

- Texel coordinates as above. Also shown for nearest filtering:
  - Given the unnormalized coordinates (u,v), the texel selected is \(ij\).
  - Given the offset \(\Delta_i\) and \(\Delta_j\), the texel selected by the offset is \(ij'\).

For corner-sampled images, the texel samples are located at the grid intersections instead of the texel centers.
16.2. Conversion Formulas

16.2.1. RGB to Shared Exponent Conversion

An RGB color (red, green, blue) is transformed to a shared exponent color (red\textsubscript{shared}, green\textsubscript{shared}, blue\textsubscript{shared}, exp\textsubscript{shared}) as follows:

First, the components (red, green, blue) are clamped to (red\textsubscript{clamped}, green\textsubscript{clamped}, blue\textsubscript{clamped}) as:

\[
\begin{align*}
\text{red}_{\text{clamped}} &= \max(0, \min(\text{sharedexp}_{\text{max}}, \text{red})) \\
\text{green}_{\text{clamped}} &= \max(0, \min(\text{sharedexp}_{\text{max}}, \text{green})) \\
\text{blue}_{\text{clamped}} &= \max(0, \min(\text{sharedexp}_{\text{max}}, \text{blue}))
\end{align*}
\]

where:

\[
\begin{align*}
N &= 9 \\
B &= 15 \\
E_{\text{max}} &= 31 \\
\text{sharedexp}_{\text{max}} &= \frac{(2^N-1)}{2^N} \times 2^{(E_{\text{max}} - B)}
\end{align*}
\]

Note

NaN, if supported, is handled as in IEEE 754-2008 \text{minNum()} and \text{maxNum()}. This results in any NaN being mapped to zero.
The largest clamped component, $\text{max}_{\text{clamped}}$ is determined:

$$
\text{max}_{\text{clamped}} = \max(\text{red}_{\text{clamped}}, \text{green}_{\text{clamped}}, \text{blue}_{\text{clamped}})
$$

A preliminary shared exponent $\exp'$ is computed:

$$
\exp' = \begin{cases} 
\lfloor \log_2(\text{max}_{\text{clamped}}) \rfloor + (B + 1) & \text{for } \text{max}_{\text{clamped}} > 2^{-(B + 1)} \\
0 & \text{for } \text{max}_{\text{clamped}} \leq 2^{-(B + 1)}
\end{cases}
$$

The shared exponent $\exp_{\text{shared}}$ is computed:

$$
\exp_{\text{shared}} = \begin{cases} 
\exp' & \text{for } 0 \leq \exp_{\text{shared}} < 2^N \\
\exp' + 1 & \text{for } \exp_{\text{shared}} = 2^N
\end{cases}
$$

Finally, three integer values in the range 0 to $2^N$ are computed:

$$
\begin{align*}
\text{red}_{\text{shared}} &= \lfloor \frac{\text{red}_{\text{clamped}}}{2^{(\exp'_{\text{shared}} - B - N)}} \rfloor + \frac{1}{2} \\
\text{green}_{\text{shared}} &= \lfloor \frac{\text{green}_{\text{clamped}}}{2^{(\exp'_{\text{shared}} - B - N)}} \rfloor + \frac{1}{2} \\
\text{blue}_{\text{shared}} &= \lfloor \frac{\text{blue}_{\text{clamped}}}{2^{(\exp'_{\text{shared}} - B - N)}} \rfloor + \frac{1}{2}
\end{align*}
$$

### 16.2.2. Shared Exponent to RGB

A shared exponent color ($\text{red}_{\text{shared}}, \text{green}_{\text{shared}}, \text{blue}_{\text{shared}}, \exp_{\text{shared}}$) is transformed to an RGB color (red, green, blue) as follows:

$$
\begin{align*}
\text{red} &= \text{red}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)} \\
\text{green} &= \text{green}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)} \\
\text{blue} &= \text{blue}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}
\end{align*}
$$

where:

$N = 9$ (number of mantissa bits per component)

$B = 15$ (exponent bias)
16.3. Texel Input Operations

Texel input instructions are SPIR-V image instructions that read from an image. Texel input operations are a set of steps that are performed on state, coordinates, and texel values while processing a texel input instruction, and which are common to some or all texel input instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Instruction/Sampler/Image validation
  - Coordinate validation
  - Sparse validation
  - Layout validation
- Format conversion
- Texel replacement
- Depth comparison
- Conversion to RGBA
- Component swizzle
- Chroma reconstruction
- $Y'CBR$ conversion

For texel input instructions involving multiple texels (for sampling or gathering), these steps are applied for each texel that is used in the instruction. Depending on the type of image instruction, other steps are conditionally performed between these steps or involving multiple coordinate or texel values.

If Chroma Reconstruction is implicit, Texel Filtering instead takes place during chroma reconstruction, before sampler $Y'CBR$ conversion occurs.

16.3.1. Texel Input Validation Operations

Texel input validation operations inspect instruction/image/sampler state or coordinates, and in certain circumstances cause the texel value to be replaced or become undefined. There are a series of validations that the texel undergoes.

Instruction/Sampler/Image View Validation

There are a number of cases where a SPIR-V instruction can mismatch with the sampler, the image view, or both, and a number of further cases where the sampler can mismatch with the image view. In such cases the value of the texel returned is undefined.

These cases include:

- The sampler borderColor is an integer type and the image view format is not one of the VkFormat integer types or a stencil component of a depth/stencil format.
• The sampler `borderColor` is a float type and the image view `format` is not one of the `VkFormat` float types or a depth component of a depth/stencil format.

• The sampler `borderColor` is one of the opaque black colors (`VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK` or `VK_BORDER_COLOR_INT_OPAQUE_BLACK`) and the image view `VkComponentSwizzle` for any of the `VkComponentMapping` components is not the `identity` swizzle.

• The sampler `borderColor` is a custom color (`VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT`) and the supplied `VkSamplerCustomBorderColorCreateInfoEXT::customBorderColor` is outside the bounds of the values representable in the image view's `format`.

• The `VkImageLayout` of any subresource in the image view does not match that specified in `VkDescriptorImageInfo::imageLayout` used to write the image descriptor.

• The SPIR-V Image Format is not `compatible` with the image view's `format`.

• The sampler `unnormalizedCoordinates` is `VK_TRUE` and any of the `limitations of unnormalized coordinates` are violated.

• The sampler was created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` and the image was not created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.

• The sampler was not created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` and the image was created with `flags` containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.

• The sampler was created with `flags` containing `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT` and is used with a function that is not `OpImageSampleImplicitLod` or `OpImageSampleExplicitLod`, or is used with operands `Offset` or `ConstOffsets`.

• The SPIR-V instruction is one of the `OpImage*Dref*` instructions and the sampler `compareEnable` is `VK_FALSE`.

• The SPIR-V instruction is not one of the `OpImage*Dref*` instructions and the sampler `compareEnable` is `VK_TRUE`.

• The SPIR-V instruction is one of the `OpImage*Dref*` instructions and the image view `format` is not one of the depth/stencil formats with a depth component, or the image view aspect is not `VK_IMAGE_ASPECT_DEPTH_BIT`.

• The SPIR-V instruction's image variable's properties are not compatible with the image view:
  ◦ Rules for `viewType`:
    ▪ `VK_IMAGE_VIEW_TYPE_1D` must have `Dim = 1D`, `Arrayed = 0`, `MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_2D` must have `Dim = 2D`, `Arrayed = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_3D` must have `Dim = 3D`, `Arrayed = 0`, `MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_CUBE` must have `Dim = Cube`, `Arrayed = 0`, `MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_1D_ARRAY` must have `Dim = 1D`, `Arrayed = 1`, `MS = 0`.
    ▪ `VK_IMAGE_VIEW_TYPE_2D_ARRAY` must have `Dim = 2D`, `Arrayed = 1`.
    ▪ `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY` must have `Dim = Cube`, `Arrayed = 1`, `MS = 0`.
  ◦ If the image was created with `VkImageCreateInfo::samples` equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction must have `MS = 0`.  

Chapter 16. Image Operations | 1077
If the image was created with `VkImageCreateInfo::samples` not equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction must have `MS = 1`.

If the Sampled Type of the OpTypeImage does not match the numeric format of the image, as shown in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table.

If the signedness of any read or sample operation does not match the signedness of the image's format.

- If the image was created with `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV`, the sampler addressing modes must only use a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

  - The SPIR-V instruction is `OpImageSampleFootprintNV` with `Dim = 2D` and `addressModeU` or `addressModeV` in the sampler is not `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

  - The SPIR-V instruction is `OpImageSampleFootprintNV` with `Dim = 3D` and `addressModeU`, `addressModeV`, or `addressModeW` in the sampler is not `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

  - The sampler was created with a specified `VkSamplerCustomBorderColorCreateInfoEXT::format` which does not match the `VkFormat` of the image view(s) it is sampling.

  - The sampler is sampling an image view of `VK_FORMAT_B4G4R4A4_UNORM_PACK16`, `VK_FORMAT_B5G6R5_UNORM_PACK16`, or `VK_FORMAT_B5G5R5A1_UNORM_PACK16` format without a specified `VkSamplerCustomBorderColorCreateInfoEXT::format`.

Only `OpImageSample*` and `OpImageSparseSample*` can be used with a sampler that enables sampler Y'CbCr conversion.

`OpImageFetch`, `OpImageSparseFetch`, `OpImage*Gather`, and `OpImageSparse*Gather` must not be used with a sampler that enables sampler Y'CbCr conversion.

The `ConstOffset` and `Offset` operands must not be used with a sampler that enables sampler Y'CbCr conversion.

**Integer Texel Coordinate Validation**

Integer texel coordinates are validated against the size of the image level, and the number of layers and number of samples in the image. For SPIR-V instructions that use integer texel coordinates, this is performed directly on the integer coordinates. For instructions that use normalized or unnormalized texel coordinates, this is performed on the coordinates that result after conversion to integer texel coordinates.

If the integer texel coordinates do not satisfy all of the conditions

\[
0 \leq i < w_s
\]

\[
0 \leq j < h_s
\]

\[
0 \leq k < d_s
\]
\[ 0 \leq l < \text{layers} \]
\[ 0 \leq n < \text{samples} \]

where:

\[ w_s = \text{width of the image level} \]

\[ h_s = \text{height of the image level} \]

\[ d_s = \text{depth of the image level} \]

\[ \text{layers} = \text{number of layers in the image} \]

\[ \text{samples} = \text{number of samples per texel in the image} \]

then the texel fails integer texel coordinate validation.

There are four cases to consider:

1. **Valid Texel Coordinates**
   - If the texel coordinates pass validation (that is, the coordinates lie within the image), then the texel value comes from the value in image memory.

2. **Border Texel**
   - If the texel coordinates fail validation, and
   - If the read is the result of an image sample instruction or image gather instruction, and
   - If the image is not a cube image,
   then the texel is a border texel and texel replacement is performed.

3. **Invalid Texel**
   - If the texel coordinates fail validation, and
   - If the read is the result of an image fetch instruction, image read instruction, or atomic instruction,
   then the texel is an invalid texel and texel replacement is performed.

4. **Cube Map Edge or Corner**
Otherwise the texel coordinates lie beyond the edges or corners of the selected cube map face, and Cube map edge handling is performed.

Cube Map Edge Handling

If the texel coordinates lie beyond the edges or corners of the selected cube map face, the following steps are performed. Note that this does not occur when using VK_FILTER_NEAREST filtering within a mip level, since VK_FILTER_NEAREST is treated as using VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE.

- Cube Map Edge Texel
  - If the texel lies beyond the selected cube map face in either only i or only j, then the coordinates (i,j) and the array layer l are transformed to select the adjacent texel from the appropriate neighboring face.

- Cube Map Corner Texel
  - If the texel lies beyond the selected cube map face in both i and j, then there is no unique neighboring face from which to read that texel. The texel should be replaced by the average of the three values of the adjacent texels in each incident face. However, implementations may replace the cube map corner texel by other methods. The methods are subject to the constraint that for linear filtering if the three available texels have the same value, the resulting filtered texel must have that value, and for cubic filtering if the twelve available samples have the same value, the resulting filtered texel must have that value.

Sparse Validation

If the texel reads from an unbound region of a sparse image, the texel is a sparse unbound texel, and processing continues with texel replacement.

Layout Validation

If all planes of a disjoint multi-planar image are not in the same image layout, the image must not be sampled with sampler Y'CbCr conversion enabled.

16.3.2. Format Conversion

Texels undergo a format conversion from the VkFormat of the image view to a vector of either floating point or signed or unsigned integer components, with the number of components based on the number of components present in the format.

- Color formats have one, two, three, or four components, according to the format.
- Depth/stencil formats are one component. The depth or stencil component is selected by the aspectMask of the image view.

Each component is converted based on its type and size (as defined in the Format Definition section for each VkFormat), using the appropriate equations in 16-Bit Floating-Point Numbers, Unsigned 11-Bit Floating-Point Numbers, Unsigned 10-Bit Floating-Point Numbers, Fixed-Point Data Conversion, and Shared Exponent to RGB. Signed integer components smaller than 32 bits are sign-extended.
If the image view format is sRGB, the color components are first converted as if they are UNORM, and then sRGB to linear conversion is applied to the R, G, and B components as described in the “sRGB EOTF” section of the Khronos Data Format Specification. The A component, if present, is unchanged.

If the image view format is block-compressed, then the texel value is first decoded, then converted based on the type and number of components defined by the compressed format.

### 16.3.3. Texel Replacement

A texel is replaced if it is one (and only one) of:

- a border texel,
- an invalid texel, or
- a sparse unbound texel.

Border texels are replaced with a value based on the image format and the `borderColor` of the sampler. The border color is:

<table>
<thead>
<tr>
<th>Sampler Border Color</th>
<th>Corresponding Border Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 0.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0.0, 0.0, 0.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE</td>
<td>([B_r, B_g, B_b, B_a] = [1.0, 1.0, 1.0, 1.0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_TRANSPARENT_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0, 0, 0, 0])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_BLACK</td>
<td>([B_r, B_g, B_b, B_a] = [0, 0, 0, 1])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT_OPAQUE_WHITE</td>
<td>([B_r, B_g, B_b, B_a] = [1, 1, 1, 1])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_FLOAT_CUSTOM_EXT</td>
<td>([B_r, B_g, B_b, B_a] = [U_r, U_g, U_b, U_a])</td>
</tr>
<tr>
<td>VK_BORDER_COLOR_INT-custom_EXT</td>
<td>([B_r, B_g, B_b, B_a] = [U_r, U_g, U_b, U_a])</td>
</tr>
</tbody>
</table>

The custom border color (U) may be rounded by implementations prior to texel replacement, but the error introduced by such a rounding must not exceed one ULP of the image’s format.

Note

The names `VK_BORDER_COLOR_\*_TRANSPARENT_BLACK`, `VK_BORDER_COLOR_\*_OPAQUE_BLACK`, and `VK_BORDER_COLOR_\*_OPAQUE_WHITE` are meant to describe which components are zeros and ones in the vocabulary of compositing, and are not meant to imply that the numerical value of `VK_BORDER_COLOR_INT_OPAQUE_WHITE` is a saturating value for integers.

This is substituted for the texel value by replacing the number of components in the image format.
<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>Component Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>( D = B_r )</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>( S = B_r )</td>
</tr>
<tr>
<td>One component color format</td>
<td>( \text{Color}_r = B_r )</td>
</tr>
<tr>
<td>Two component color format</td>
<td>( [\text{Color}_r, \text{Color}_g] = [B_r, B_g] )</td>
</tr>
<tr>
<td>Three component color format</td>
<td>( [\text{Color}_r, \text{Color}_g, \text{Color}_b] = [B_r, B_g, B_b] )</td>
</tr>
<tr>
<td>Four component color format</td>
<td>( [\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a] = [B_r, B_g, B_b, B_a] )</td>
</tr>
</tbody>
</table>

The value returned by a read of an invalid texel is undefined, unless that read operation is from a buffer resource and the robustBufferAccess feature is enabled. In that case, an invalid texel is replaced as described by the robustBufferAccess feature. If the access is to an image resource and the x, y, z, or layer coordinate validation fails and robustImageAccess is enabled then zero must be returned for the R, G, and B channels, if present. Either zero or one must be returned for the A channel, if present. If robustImageAccess2 is enabled, zero values must be returned. If only the sample index was invalid, the values returned are undefined.

Additionally, if robustImageAccess is enabled, but robustImageAccess2 is not, any invalid texels may be expanded to four components prior to texel replacement. This means that components not present in the image format may be replaced with 0 or may undergo conversion to RGBA as normal.

Loads from a null descriptor return a four component color value of all zeros. However, for storage images and storage texel buffers using an explicit SPIR-V Image Format, loads from a null descriptor may return an alpha value of 1 (float or integer, depending on format) if the format doesn’t include alpha.

If the VkPhysicalDeviceSparseProperties::residencyNonResidentStrict property is VK_TRUE, a sparse unbound texel is replaced with 0 or 0.0 values for integer and floating-point components of the image format, respectively.

If residencyNonResidentStrict is VK_FALSE, the value of the sparse unbound texel is undefined.

### 16.3.4. Depth Compare Operation

If the image view has a depth/stencil format, the depth component is selected by the aspectMask, and the operation is a Dref instruction, a depth comparison is performed. The value of the result D is 1.0 if the result of the compare operation is true, and 0.0 otherwise. The compare operation is selected by the compareOp member of the sampler.

\[
D = \begin{cases} 
1.0 & \text{for LEQUAL} \\
0.0 & \text{for GEQUAL} \\
D_{\text{ref}} < D_{\text{tex}} & \text{for LESS} \\
D_{\text{ref}} \geq D_{\text{tex}} & \text{for GREATER} \\
D_{\text{ref}} = D_{\text{tex}} & \text{for EQUAL} \\
D_{\text{ref}} \neq D_{\text{tex}} & \text{for NOTEQUAL} \\
\text{true} & \text{for ALWAYS} \\
\text{false} & \text{for NEVER}
\end{cases}
\]

1082 | Chapter 16. Image Operations
where \( D_{\text{tex}} \) is the texel depth value and \( D_{\text{ref}} \) is the reference value from the SPIR-V operand. If the image being sampled has a fixed-point format then the reference value is clamped to \([0, 1]\) before the comparison operation.

### 16.3.5. Conversion to RGBA

The texel is expanded from one, two, or three components to four components based on the image base color:

*Table 26. Texel Color After Conversion To RGBA*

<table>
<thead>
<tr>
<th>Texel Aspect or Format</th>
<th>RGBA Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth aspect</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a]) = ([D,0,0,\text{one}])</td>
</tr>
<tr>
<td>Stencil aspect</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a]) = ([S,0,0,\text{one}])</td>
</tr>
<tr>
<td>One component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a]) = ([\text{Color}_r,0,0,\text{one}])</td>
</tr>
<tr>
<td>Two component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a]) = ([\text{Color}_r,\text{Color}_g,0,\text{one}])</td>
</tr>
<tr>
<td>Three component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a]) = ([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{one}])</td>
</tr>
<tr>
<td>Four component color format</td>
<td>([\text{Color}_r, \text{Color}_g, \text{Color}_b, \text{Color}_a]) = ([\text{Color}_r,\text{Color}_g,\text{Color}_b,\text{Color}_a])</td>
</tr>
</tbody>
</table>

where \( \text{one} = 1.0f \) for floating-point formats and depth aspects, and \( \text{one} = 1 \) for integer formats and stencil aspects.

### 16.3.6. Component Swizzle

All texel input instructions apply a *swizzle* based on:

- the \text{VkComponentSwizzle} enums in the \text{components} member of the \text{VkImageViewCreateInfo} structure for the image being read if sampler \text{Y'CrC} conversion is not enabled, and
- the \text{VkComponentSwizzle} enums in the \text{components} member of the \text{VkSamplerYcbcrConversionCreateInfo} structure for the sampler \text{Y'CrC} conversion if sampler \text{Y'CrC} conversion is enabled.

The swizzle can rearrange the components of the texel, or substitute zero or one for any components. It is defined as follows for each color component:

\[
\text{Color}'_{\text{component}} = \begin{cases} 
\text{Color}_r & \text{for RED swizzle} \\
\text{Color}_g & \text{for GREEN swizzle} \\
\text{Color}_b & \text{for BLUE swizzle} \\
\text{Color}_a & \text{for ALPHA swizzle} \\
0 & \text{for ZERO swizzle} \\
\text{one} & \text{for ONE swizzle} \\
\text{identity} & \text{for IDENTITY swizzle}
\end{cases}
\]

where:
If the border color is one of the `VK_BORDER_COLOR_*_OPAQUE_BLACK` enums and the `VkComponentSwizzle` is not the identity swizzle for all components, the value of the texel after swizzle is undefined.

### 16.3.7. Sparse Residency

`OpImageSparse` instructions return a structure which includes a *residency code* indicating whether any texels accessed by the instruction are sparse unbound texels. This code can be interpreted by the `OpImageSparseTexelsResident` instruction which converts the residency code to a boolean value.

### 16.3.8. Chroma Reconstruction

In some color models, the color representation is defined in terms of monochromatic light intensity (often called “luma”) and color differences relative to this intensity, often called “chroma”. It is common for color models other than RGB to represent the chroma channels at lower spatial resolution than the luma channel. This approach is used to take advantage of the eye’s lower spatial sensitivity to color compared with its sensitivity to brightness. Less commonly, the same approach is used with additive color, since the green channel dominates the eye’s sensitivity to light intensity and the spatial sensitivity to color introduced by red and blue is lower.

Lower-resolution channels are “downsampled” by resizing them to a lower spatial resolution than the channel representing luminance. This process is also commonly known as “chroma subsampling”. There is one luminance sample in each texture texel, but each chrominance sample may be shared among several texels in one or both texture dimensions.

- “.444” formats do not spatially downsample chroma values compared with luma: there are unique chroma samples for each texel.
- “.422” formats have downsampling in the x dimension (corresponding to \(u\) or \(s\) coordinates): they are sampled at half the resolution of luma in that dimension.
- “.420” formats have downsampling in the x dimension (corresponding to \(u\) or \(s\) coordinates) and the y dimension (corresponding to \(v\) or \(t\) coordinates): they are sampled at half the resolution of luma in both dimensions.

The process of reconstructing a full color value for texture access involves accessing both chroma and luma values at the same location. To generate the color accurately, the values of the lower-resolution channels at the location of the luma samples must be reconstructed from the lower-resolution sample locations, an operation known here as “chroma reconstruction” irrespective of the actual color model.

The location of the chroma samples relative to the luma coordinates is determined by the `xChromaOffset` and `yChromaOffset` members of the `VkSamplerYcbcrConversionCreateInfo` structure.
used to create the sampler $Y'C_bC_r$ conversion.

The following diagrams show the relationship between unnormalized $(u,v)$ coordinates and $(i,j)$ integer texel positions in the luma channel (shown in black, with circles showing integer sample positions) and the texel coordinates of reduced-resolution chroma channels, shown as crosses in red.

**Note**

If the chroma values are reconstructed at the locations of the luma samples by means of interpolation, chroma samples from outside the image bounds are needed; these are determined according to Wrapping Operation. These diagrams represent this by showing the bounds of the “chroma texel” extending beyond the image bounds, and including additional chroma sample positions where required for interpolation. The limits of a sample for NEAREST sampling is shown as a grid.

![Diagram showing the relationship between unnormalized coordinates and integer texel positions](image)

*Figure 6. 4:2:2 downsampling, xChromaOffset=COSITED_EVEN*
Figure 7. 4:2:2 downsampling, xChromaOffset=MIDPOINT

Figure 8. 4:2:0 downsampling, xChromaOffset=COSITED_EVEN, yChromaOffset=COSITED_EVEN
Figure 9. 420 downsampling, xChromaOffset=MIDPOINT, yChromaOffset=COSITED_EVEN

Figure 10. 420 downsampling, xChromaOffset=COSITED_EVEN, yChromaOffset=MIDPOINT
Reconstruction is implemented in one of two ways:

If the format of the image that is to be sampled sets `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT`, or the `VkSamplerYcbcrConversionCreateInfo`'s `forceExplicitReconstruction` is set to `VK_TRUE`, reconstruction is performed as an explicit step independent of filtering, described in the Explicit Reconstruction section.

If the format of the image that is to be sampled does not set `VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT` and if the `VkSamplerYcbcrConversionCreateInfo`'s `forceExplicitReconstruction` is set to `VK_FALSE`, reconstruction is performed as an implicit part of filtering prior to color model conversion, with no separate post-conversion texel filtering step, as described in the Implicit Reconstruction section.

Explicit Reconstruction

• If the `chromaFilter` member of the `VkSamplerYcbcrConversionCreateInfo` structure is `VK_FILTER_NEAREST`:
  
  ◦ If the format's R and B channels are reduced in resolution in just width by a factor of two relative to the G channel (i.e. this is a “._422” format), the $\tau_{ijk}[\text{level}]$ values accessed by texel filtering are reconstructed as follows:

    $\tau_{R^{'}}(i, j) = \tau_{R}(i \times 0.5, j)[\text{level}]$

    $\tau_{B^{'}}(i, j) = \tau_{B}(i \times 0.5, j)[\text{level}]$

  ◦ If the format's R and B channels are reduced in resolution in width and height by a factor of two relative to the G channel (i.e. this is a “._420” format), the $\tau_{ijk}[\text{level}]$ values accessed by
texel filtering are reconstructed as follows:

\[
\begin{align*}
\tau_R'(i, j) &= \tau_R([i \times 0.5], [j \times 0.5]) \text{[level]} \\
\tau_B'(i, j) &= \tau_B([i \times 0.5], [j \times 0.5]) \text{[level]}
\end{align*}
\]

**Note**

\(x\text{ChromaOffset}\) and \(y\text{ChromaOffset}\) have no effect if \(\text{chromaFilter}\) is \(\text{VK}\_\text{FILTER}\_\text{NEAREST}\) for explicit reconstruction.

- If the \(\text{chromaFilter}\) member of the \(\text{VkSamplerYcbcrConversionCreateInfo}\) structure is \(\text{VK}\_\text{FILTER}\_\text{LINEAR}\):
  - If the format’s R and B channels are reduced in resolution in just width by a factor of two relative to the G channel (i.e. this is a “\_422” format):
    - If \(x\text{ChromaOffset}\) is \(\text{VK}\_\text{CHROMA}\_\text{LOCATION}\_\text{COSITED}\_\text{EVEN}):
      \[
      \tau_{RB}'(i, j) = \begin{cases} 
      \tau_{RB}([i \times 0.5], j) \text{[level]}, & 0.5 \times i = [0.5 \times i] \\
      0.5 \times \tau_{RB}([i \times 0.5], j) \text{[level]} + 0.5 \times \tau_{RB}([i \times 0.5] + 1, j) \text{[level]}, & 0.5 \times i \neq [0.5 \times i]
      \end{cases}
      \]
    - If \(x\text{ChromaOffset}\) is \(\text{VK}\_\text{CHROMA}\_\text{LOCATION}\_\text{MIDPOINT}):
      \[
      \tau_{RB}'(i, j) = \begin{cases} 
      0.25 \times \tau_{RB}([i \times 0.5] - 1, j) \text{[level]} + 0.75 \times \tau_{RB}([i \times 0.5], j) \text{[level]}, & 0.5 \times i = [0.5 \times i] \\
      0.75 \times \tau_{RB}([i \times 0.5] - 1, j) \text{[level]} + 0.25 \times \tau_{RB}([i \times 0.5] + 1, j) \text{[level]}, & 0.5 \times i \neq [0.5 \times i]
      \end{cases}
      \]
  - If the format’s R and B channels are reduced in resolution in width and height by a factor of two relative to the G channel (i.e. this is a “\_420” format), a similar relationship applies. Due to the number of options, these formulae are expressed more concisely as follows:

\[
\begin{align*}
i_{RB} &= \begin{cases} 
0.5 \times (i) & \text{xChromaOffset=}\text{COSITED}\_\text{EVEN} \\
0.5 \times (i - 0.5) & \text{xChromaOffset=}\text{MIDPOINT}
\end{cases} \\
j_{RB} &= \begin{cases} 
0.5 \times (j) & \text{yChromaOffset=}\text{COSITED}\_\text{EVEN} \\
0.5 \times (j - 0.5) & \text{yChromaOffset=}\text{MIDPOINT}
\end{cases}
\end{align*}
\]

\[
i_{\text{floor}} = \lfloor i_{RB} \rfloor \\
j_{\text{floor}} = \lfloor j_{RB} \rfloor \\
i_{\text{frac}} = i_{RB} - i_{\text{floor}} \\
j_{\text{frac}} = j_{RB} - j_{\text{floor}}
\]

\[
\begin{align*}
\tau_{RB}'(i, j) &= \tau_{RB}(i_{\text{floor}}, j_{\text{floor}}) \text{[level]} \\
&\times (1 - i_{\text{frac}}) \times (1 - j_{\text{frac}}) \\
\tau_{RB}(1 + i_{\text{floor}}, j_{\text{floor}}) \text{[level]} &= \tau_{RB}(1 + i_{\text{floor}}, j_{\text{floor}}) \text{[level]} \\
&\times (i_{\text{frac}}) \times (1 - j_{\text{frac}}) \\
\tau_{RB}(i_{\text{floor}}, 1 + j_{\text{floor}}) \text{[level]} &= \tau_{RB}(i_{\text{floor}}, 1 + j_{\text{floor}}) \text{[level]} \\
&\times (1 - i_{\text{frac}}) \times (j_{\text{frac}}) \\
\tau_{RB}(1 + i_{\text{floor}}, 1 + j_{\text{floor}}) \text{[level]} &= \tau_{RB}(1 + i_{\text{floor}}, 1 + j_{\text{floor}}) \text{[level]} \\
&\times (i_{\text{frac}}) \times (j_{\text{frac}})
\end{align*}
\]
Note
In the case where the texture itself is bilinearly interpolated as described in Texel Filtering, thus requiring four full-color samples for the filtering operation, and where the reconstruction of these samples uses bilinear interpolation in the chroma channels due to `chromaFilter=VK_FILTER_LINEAR`, up to nine chroma samples may be required, depending on the sample location.

Implicit Reconstruction

Implicit reconstruction takes place by the samples being interpolated, as required by the filter settings of the sampler, except that `chromaFilter` takes precedence for the chroma samples.

If `chromaFilter` is `VK_FILTER_NEAREST`, an implementation may behave as if `xChromaOffset` and `yChromaOffset` were both `VK_CHROMA_LOCATION_MIDPOINT`, irrespective of the values set.

Note
This will not have any visible effect if the locations of the luma samples coincide with the location of the samples used for rasterization.

The sample coordinates are adjusted by the downsample factor of the channel (such that, for example, the sample coordinates are divided by two if the channel has a downsample factor of two relative to the luma channel):

\[
\begin{align*}
    u_{RB}'(422/420) &= \left\{ 
    \begin{array}{ll}
    0.5 \times (u + 0.5), & xChromaOffset=\text{COSITED\_EVEN} \\
    0.5 \times u, & xChromaOffset=\text{MIDPOINT} 
    \end{array}
    \right.
    \\
    v_{RB}'(422) &= \left\{ 
    \begin{array}{ll}
    0.5 \times (v + 0.5), & yChromaOffset=\text{COSITED\_EVEN} \\
    0.5 \times v, & yChromaOffset=\text{MIDPOINT} 
    \end{array}
    \right.
\end{align*}
\]

16.3.9. Sampler Y’C_bC_r Conversion

Sampler Y’C_bC_r conversion performs the following operations, which an implementation may combine into a single mathematical operation:

- Sampler Y’C_bC_r Range Expansion
- Sampler Y’C_bC_r Model Conversion

Sampler Y’C_bC_r Range Expansion

Sampler Y’C_bC_r range expansion is applied to color channel values after all texel input operations which are not specific to sampler Y’C_bC_r conversion. For example, the input values to this stage have been converted using the normal format conversion rules.

Sampler Y’C_bC_r range expansion is not applied if `ycbcrModel` is `VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY`. That is, the shader receives the vector C’rgba as output by the Component Swizzle stage without further modification.

For other values of `ycbcrModel`, range expansion is applied to the texel channel values output by the Component Swizzle defined by the `components` member of `VkSamplerYcbcrConversionCreateInfo`.
Range expansion applies independently to each channel of the image. For the purposes of range expansion and Y’C₈C₉ model conversion, the R and B channels contain color difference (chroma) values and the G channel contains luma. The A channel is not modified by sampler Y’C₈C₉ range expansion.

The range expansion to be applied is defined by the `ycbcrRange` member of the `VkSamplerYcbcrConversionCreateInfo` structure:

- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_FULL`, the following transformations are applied:

  \[
  Y' = C'_\text{gba} [G] \\
  C_B = C'_\text{gba} [B] - \frac{2^{(n-1)}}{(2^n - 1)} \\
  C_R = C'_\text{gba} [R] - \frac{2^{(n-1)}}{(2^n - 1)}
  \]

  *Note*

  These formulae correspond to the “full range” encoding in the “Quantization schemes” chapter of the Khronos Data Format Specification.

  Should any future amendments be made to the ITU specifications from which these equations are derived, the formulae used by Vulkan may also be updated to maintain parity.

- If `ycbcrRange` is `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW`, the following transformations are applied:

  \[
  Y' = C'_\text{gba} [G] \times (2^n - 1) - 16 \times 2^n - 8 \\
  C_B = C'_\text{gba} [B] \times (2^n - 1) - 128 \times 2^n - 8 \\
  C_R = C'_\text{gba} [R] \times (2^n - 1) - 128 \times 2^n - 8 \\
  \]

  *Note*

  These formulae correspond to the “narrow range” encoding in the “Quantization schemes” chapter of the Khronos Data Format Specification.

- \(n\) is the bit-depth of the channels in the format.

The precision of the operations performed during range expansion must be at least that of the source format.

An implementation may clamp the results of these range expansion operations such that \(Y'\) falls in the range \([0,1]\), and/or such that \(C_B\) and \(C_R\) fall in the range \([-0.5,0.5]\).

**Sampler Y’C₈C₉ Model Conversion**

The range-expanded values are converted between color models, according to the color model conversion specified in the `ycbcrModel` member:
VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY

The color channels are not modified by the color model conversion since they are assumed already to represent the desired color model in which the shader is operating; \( Y'C_bC_R \) range expansion is also ignored.

VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY

The color channels are not modified by the color model conversion and are assumed to be treated as though in \( Y'C_bC_R \) form both in memory and in the shader; \( Y'C_bC_R \) range expansion is applied to the channels as for other \( Y'C_bC_R \) models, with the vector \((C_R,Y',C_B,A)\) provided to the shader.

VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709

The color channels are transformed from a \( Y'C_bC_R \) representation to an \( R'G'B' \) representation as described in the “BT.709 \( Y'C_bC_R \) conversion” section of the Khronos Data Format Specification.

VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601

The color channels are transformed from a \( Y'C_bC_R \) representation to an \( R'G'B' \) representation as described in the “BT.601 \( Y'C_bC_R \) conversion” section of the Khronos Data Format Specification.

VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020

The color channels are transformed from a \( Y'C_bC_R \) representation to an \( R'G'B' \) representation as described in the “BT.2020 \( Y'C_bC_R \) conversion” section of the Khronos Data Format Specification.

In this operation, each output channel is dependent on each input channel.

An implementation may clamp the \( R'G'B' \) results of these conversions to the range \([0,1]\).

The precision of the operations performed during model conversion must be at least that of the source format.

The alpha channel is not modified by these model conversions.
Note

Sampling operations in a non-linear color space can introduce color and intensity shifts at sharp transition boundaries. To avoid this issue, the technically precise color correction sequence described in the “Introduction to Color Conversions” chapter of the Khronos Data Format Specification may be performed as follows:

- Calculate the unnormalized texel coordinates corresponding to the desired sample position.
- For a minFilter or magFilter of VK_FILTER_NEAREST:
  1. Calculate \((i, j)\) for the sample location as described under the “nearest filtering” formulae in \((u, v, w, a)\) to \((i, j, k, l, n)\) Transformation And Array Layer Selection
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using sampler Y'C_bC_r conversion at this location.
- For a minFilter or magFilter of VK_FILTER_LINEAR:
  1. Calculate \((i_{0,1}, j_{0,1})\) for the sample location as described under the “linear filtering” formulae in \((u, v, w, a)\) to \((i, j, k, l, n)\) Transformation And Array Layer Selection
  2. Calculate the normalized texel coordinates corresponding to these integer coordinates.
  3. Sample using sampler Y'C_bC_r conversion at each of these locations.
  4. Convert the non-linear A'R'G'B' outputs of the Y'C_bC_r conversions to linear ARGB values as described in the “Transfer Functions” chapter of the Khronos Data Format Specification.
  5. Interpolate the linear ARGB values using the \(\alpha\) and \(\beta\) values described in the “linear filtering” section of \((u, v, w, a)\) to \((i, j, k, l, n)\) Transformation And Array Layer Selection and the equations in Texel Filtering.

The additional calculations and, especially, additional number of sampling operations in the VK_FILTER_LINEAR case can be expected to have a performance impact compared with using the outputs directly. Since the variations from “correct” results are subtle for most content, the application author should determine whether a more costly implementation is strictly necessary.

If chromaFilter, and minFilter or magFilter are both VK_FILTER_NEAREST, these operations are redundant and sampling using sampler Y'C_bC_r conversion at the desired sample coordinates will produce the “correct” results without further processing.

16.4. Texel Output Operations

Texel output instructions are SPIR-V image instructions that write to an image. Texel output
operations are a set of steps that are performed on state, coordinates, and texel values while processing a texel output instruction, and which are common to some or all texel output instructions. They include the following steps, which are performed in the listed order:

- **Validation operations**
  - Format validation
  - Type validation
  - Coordinate validation
  - Sparse validation
- **Texel output format conversion**

### 16.4.1. Texel Output Validation Operations

*Texel output validation operations* inspect instruction/image state or coordinates, and in certain circumstances cause the write to have no effect. There are a series of validations that the texel undergoes.

**Texel Format Validation**

If the image format of the `OpTypeImage` is not compatible with the `VkImageView`'s format, the write causes the contents of the image's memory to become undefined.

**Texel Type Validation**

If the Sampled Type of the `OpTypeImage` does not match the type defined for the format, as specified in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table, the write causes the value of the texel to become undefined. For integer types, if the Signedness of the Sampled Type of the `OpTypeImage` does not match the signedness of the accessed resource, the write causes the value of the texel to become undefined.

### 16.4.2. Integer Texel Coordinate Validation

The integer texel coordinates are validated according to the same rules as for texel input coordinate validation.

If the texel fails integer texel coordinate validation, then the write has no effect.

### 16.4.3. Sparse Texel Operation

If the texel attempts to write to an unbound region of a sparse image, the texel is a sparse unbound texel. In such a case, if the `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` property is `VK_TRUE`, the sparse unbound texel write has no effect. If `residencyNonResidentStrict` is `VK_FALSE`, the write may have a side effect that becomes visible to other accesses to unbound texels in any resource, but will not be visible to any device memory allocated by the application.
16.4.4. Texel Output Format Conversion

If the image format is sRGB, a linear to sRGB conversion is applied to the R, G, and B components as described in the “sRGB EOTF” section of the Khronos Data Format Specification. The A component, if present, is unchanged.

Texels then undergo a format conversion from the floating point, signed, or unsigned integer type of the texel data to the VkFormat of the image view. Any unused components are ignored.

Each component is converted based on its type and size (as defined in the Format Definition section for each VkFormat). Floating-point outputs are converted as described in Floating-Point Format Conversions and Fixed-Point Data Conversion. Integer outputs are converted such that their value is preserved. The converted value of any integer that cannot be represented in the target format is undefined.

16.5. Normalized Texel Coordinate Operations

If the image sampler instruction provides normalized texel coordinates, some of the following operations are performed.

16.5.1. Projection Operation

For Proj image operations, the normalized texel coordinates \((s, t, r, q, a)\) and (if present) the \(D_{\text{ref}}\) coordinate are transformed as follows:

\[
\begin{align*}
    s &= \frac{s}{q}, & \text{for 1D, 2D, or 3D image} \\
    t &= \frac{t}{q}, & \text{for 2D or 3D image} \\
    r &= \frac{r}{q}, & \text{for 3D image} \\
    D_{\text{ref}} &= \frac{D_{\text{ref}}}{q}, & \text{if provided}
\end{align*}
\]

16.5.2. Derivative Image Operations

Derivatives are used for LOD selection. These derivatives are either implicit (in an ImplicitLod image instruction in a fragment shader) or explicit (provided explicitly by shader to the image instruction in any shader).

For implicit derivatives image instructions, the derivatives of texel coordinates are calculated in the same manner as derivative operations. That is:

\[
\begin{align*}
    \frac{\partial s}{\partial x} &= dPdx(s), & \frac{\partial s}{\partial y} &= dPdy(s), & \text{for 1D, 2D, Cube, or 3D image} \\
    \frac{\partial t}{\partial x} &= dPdx(t), & \frac{\partial t}{\partial y} &= dPdy(t), & \text{for 2D, Cube, or 3D image} \\
    \frac{\partial r}{\partial x} &= dPdx(r), & \frac{\partial r}{\partial y} &= dPdy(r), & \text{for Cube or 3D image}
\end{align*}
\]

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit LOD image instructions, if the optional SPIR-V operand Grad is provided, then the operand values are used for the derivatives. The number of components present in each derivative
for a given image dimensionality matches the number of partial derivatives computed above.

If the optional SPIR-V operand \( \text{Lod} \) is provided, then derivatives are set to zero, the cube map derivative transformation is skipped, and the scale factor operation is skipped. Instead, the floating point scalar coordinate is directly assigned to \( \lambda_{\text{base}} \) as described in Level-of-Detail Operation.

If the image or sampler object used by an implicit derivative image instruction is not uniform across the quad and \( \text{quadDivergentImplicitLod} \) is not supported, then the derivative and LOD values are undefined. Implicit derivatives are well-defined when the image and sampler and control flow are uniform across the quad, even if they diverge between different quads.

If \( \text{quadDivergentImplicitLod} \) is supported, then derivatives and implicit LOD values are well-defined even if the image or sampler object are not uniform within a quad. The derivatives are computed as specified above, and the implicit LOD calculation proceeds for each shader invocation using its respective image and sampler object.

### 16.5.3. Cube Map Face Selection and Transformations

For cube map image instructions, the \((s,t,r)\) coordinates are treated as a direction vector \((r_x,r_y,r_z)\). The direction vector is used to select a cube map face. The direction vector is transformed to a per-face texel coordinate system \((s\text{face},t\text{face})\), The direction vector is also used to transform the derivatives to per-face derivatives.

### 16.5.4. Cube Map Face Selection

The direction vector selects one of the cube map’s faces based on the largest magnitude coordinate direction (the major axis direction). Since two or more coordinates can have identical magnitude, the implementation must have rules to disambiguate this situation.

The rules should have as the first rule that \( r_z \) wins over \( r_y \) and \( r_x \), and the second rule that \( r_y \) wins over \( r_x \). An implementation may choose other rules, but the rules must be deterministic and depend only on \((r_x,r_y,r_z)\).

The layer number (corresponding to a cube map face), the coordinate selections for \( s_c, t_c, r_c \), and the selection of derivatives, are determined by the major axis direction as specified in the following two tables.

#### Table 27. Cube map face and coordinate selection

<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>Layer Number</th>
<th>Cube Map Face</th>
<th>( s_c )</th>
<th>( t_c )</th>
<th>( r_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+r_x)</td>
<td>0</td>
<td>Positive X</td>
<td>(-r_z)</td>
<td>(-r_y)</td>
<td>( r_x)</td>
</tr>
<tr>
<td>(-r_x)</td>
<td>1</td>
<td>Negative X</td>
<td>(+r_z)</td>
<td>(-r_y)</td>
<td>( r_x)</td>
</tr>
<tr>
<td>(+r_y)</td>
<td>2</td>
<td>Positive Y</td>
<td>(+r_x)</td>
<td>(+r_z)</td>
<td>( r_y)</td>
</tr>
<tr>
<td>(-r_y)</td>
<td>3</td>
<td>Negative Y</td>
<td>(+r_x)</td>
<td>(-r_z)</td>
<td>( r_y)</td>
</tr>
<tr>
<td>(+r_z)</td>
<td>4</td>
<td>Positive Z</td>
<td>(+r_x)</td>
<td>(-r_y)</td>
<td>( r_z)</td>
</tr>
</tbody>
</table>
16.5.5. Cube Map Coordinate Transformation

\[ s_{face} = \frac{1}{2} \times \frac{s_c}{r_c} + \frac{1}{2} \]

\[ t_{face} = \frac{1}{2} \times \frac{t_c}{r_c} + \frac{1}{2} \]

16.5.6. Cube Map Derivative Transformation

\[ \frac{\partial s_{face}}{\partial x} = \frac{\partial}{\partial x} \left( \frac{1}{2} \times \frac{s_c}{r_c} + \frac{1}{2} \right) \]

\[ = \frac{1}{2} \times \frac{\partial \left( \frac{s_c}{r_c} \right)}{\partial x} \]

\[ = \frac{1}{2} \times \left( \frac{r_c \times \partial s_c / \partial x - s_c \times \partial r_c / \partial x}{(r_c)^2} \right) \]

\[ \frac{\partial s_{face}}{\partial y} = \frac{\partial}{\partial y} \left( \frac{1}{2} \times \frac{s_c}{r_c} + \frac{1}{2} \right) \]

\[ = \frac{1}{2} \times \left( \frac{r_c \times \partial s_c / \partial y - s_c \times \partial r_c / \partial y}{(r_c)^2} \right) \]

16.5.7. Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection

LOD selection can be either explicit (provided explicitly by the image instruction) or implicit (determined from a scale factor calculated from the derivatives). The LOD must be computed with \texttt{mipmapPrecisionBits} of accuracy.
Scale Factor Operation

The magnitude of the derivatives are calculated by:

\[
\begin{align*}
    m_{ux} &= |\partial s/\partial x| \times w_{\text{base}} \\
    m_{vx} &= |\partial t/\partial x| \times h_{\text{base}} \\
    m_{wx} &= |\partial r/\partial x| \times d_{\text{base}} \\
    m_{uy} &= |\partial s/\partial y| \times w_{\text{base}} \\
    m_{vy} &= |\partial t/\partial y| \times h_{\text{base}} \\
    m_{wy} &= |\partial r/\partial y| \times d_{\text{base}}
\end{align*}
\]

where:

\[
\begin{align*}
    \partial t/\partial x = \partial t/\partial y &= 0 \text{ (for 1D images)} \\
    \partial r/\partial x = \partial r/\partial y &= 0 \text{ (for 1D, 2D or Cube images)}
\end{align*}
\]

and:

\[
\begin{align*}
    w_{\text{base}} &= \text{image.w} \\
    h_{\text{base}} &= \text{image.h} \\
    d_{\text{base}} &= \text{image.d}
\end{align*}
\]

(for the baseMipLevel, from the image descriptor).

For corner-sampled images, the \(w_{\text{base}}, h_{\text{base}},\) and \(d_{\text{base}}\) are instead:

\[
\begin{align*}
    w_{\text{base}} &= \text{image.w} - 1 \\
    h_{\text{base}} &= \text{image.h} - 1
\end{align*}
\]
\[ d_{\text{base}} = \text{image.d} - 1 \]

A point sampled in screen space has an elliptical footprint in texture space. The minimum and maximum scale factors \((\rho_{\text{min}}, \rho_{\text{max}})\) should be the minor and major axes of this ellipse.

The scale factors \(\rho_x\) and \(\rho_y\), calculated from the magnitude of the derivatives in \(x\) and \(y\), are used to compute the minimum and maximum scale factors.

\(\rho_x\) and \(\rho_y\) may be approximated with functions \(f_x\) and \(f_y\), subject to the following constraints:

\[
\begin{align*}
    & f_x \text{ is continuous and monotonically increasing in each of } m_{ux}, m_{vx}, \text{ and } m_{wx} \\
    & f_y \text{ is continuous and monotonically increasing in each of } m_{uy}, m_{vy}, \text{ and } m_{wy} \\
    & \max(|m_{ux}|, |m_{vx}|, |m_{wx}|) \leq f_x \leq \sqrt{2}(|m_{ux}| + |m_{vx}| + |m_{wx}|) \\
    & \max(|m_{uy}|, |m_{vy}|, |m_{wy}|) \leq f_y \leq \sqrt{2}(|m_{uy}| + |m_{vy}| + |m_{wy}|)
\end{align*}
\]

The minimum and maximum scale factors \((\rho_{\text{min}}, \rho_{\text{max}})\) are determined by:

\[
\begin{align*}
    \rho_{\text{max}} &= \max(\rho_x, \rho_y) \\
    \rho_{\text{min}} &= \min(\rho_x, \rho_y)
\end{align*}
\]

The ratio of anisotropy is determined by:

\[
\eta = \min(\rho_{\text{max}}/\rho_{\text{min}}, \max_{\text{Aniso}})
\]

where:

\[
\text{sampler.max}_{\text{Aniso}} = \max_{\text{Anisotropy}} \text{ (from sampler descriptor)}
\]

\[
\text{limits.max}_{\text{Aniso}} = \max_{\text{SamplerAnisotropy}} \text{ (from physical device limits)}
\]

\[
\max_{\text{Aniso}} = \min(\text{sampler.max}_{\text{Aniso}}, \text{limits.max}_{\text{Aniso}})
\]

If \(\rho_{\text{max}} = \rho_{\text{min}} = 0\), then all the partial derivatives are zero, the fragment’s footprint in texel space is a point, and \(\eta\) should be treated as 1. If \(\rho_{\text{max}} \neq 0\) and \(\rho_{\text{min}} = 0\) then all partial derivatives along one axis are zero, the fragment’s footprint in texel space is a line segment, and \(\eta\) should be treated as \(\max_{\text{Aniso}}\). However, anytime the footprint is small in texel space the implementation may use a smaller value of \(\eta\), even when \(\rho_{\text{min}}\) is zero or close to zero. If either \(\text{VkPhysicalDeviceFeatures::samplerAnisotropy}\) or \(\text{VkSamplerCreateInfo::anisotropyEnable}\) are \(\text{VK_FALSE}\), \(\max_{\text{Aniso}}\) is set to 1.

If \(\eta = 1\), sampling is isotropic. If \(\eta > 1\), sampling is anisotropic.
The sampling rate \( N \) is derived as:

\[
N = \lceil \eta \rceil
\]

An implementation may round \( N \) up to the nearest supported sampling rate. An implementation may use the value of \( N \) as an approximation of \( \eta \).

**Level-of-Detail Operation**

The LOD parameter \( \lambda \) is computed as follows:

\[
\lambda_{\text{base}}(x, y) = \begin{cases} 
\text{shaderOp.Lod} & \text{(from optional SPIR-V operand)} \\
\log_2 \left( \frac{P_{\text{max}}}{\eta} \right) & \text{otherwise}
\end{cases}
\]

\[
\lambda'(x, y) = \lambda_{\text{base}} + \text{clamp}(\text{sampler.bias} + \text{shaderOp.bias}, -\text{maxSamplerLodBias}, \text{maxSamplerLodBias})
\]

\[
\lambda = \begin{cases} 
\text{lod}_{\text{max}}, & \lambda' > \text{lod}_{\text{max}} \\
\lambda', & \text{lod}_{\text{min}} \leq \lambda' \leq \text{lod}_{\text{max}} \\
\text{lod}_{\text{min}}, & \lambda' < \text{lod}_{\text{min}} \\
\text{undefined}, & \text{lod}_{\text{min}} > \text{lod}_{\text{max}}
\end{cases}
\]

where:

\[
\text{sampler.bias} = \text{mipLodBias}
\]

\[
\text{shaderOp.bias} = \begin{cases} 
\text{Bias} & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{sampler.lod}_{\text{min}} = \text{minLod}
\]

\[
\text{shaderOp.lod}_{\text{min}} = \begin{cases} 
\text{MinLod} & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{lod}_{\text{min}} = \max(\text{sampler.lod}_{\text{min}}, \text{shaderOp.lod}_{\text{min}})
\]

\[
\text{lod}_{\text{max}} = \maxLod
\]

and \text{maxSamplerLodBias} is the value of the \text{VkPhysicalDeviceLimits} feature \text{maxSamplerLodBias}.

**Image Level(s) Selection**

The image level(s) \( d, d_{\text{hi}}, \text{and } d_{\text{lo}} \) which texels are read from are determined by an image-level parameter \( d_l \), which is computed based on the LOD parameter, as follows:

\[
d_l = \begin{cases} 
\text{nearest}(d'), & \text{mipmapMode is VK_SAMPLER_MIPMAP_MODE_NEAREST} \\
d', & \text{otherwise}
\end{cases}
\]

where:

\[
d' = \text{level}_{\text{base}} + \text{clamp}(\lambda, 0, q)
\]

\[
\text{nearest}(d') = \begin{cases} 
[d' + 0.5] - 1, & \text{preferred} \\
[d' + 0.5], & \text{alternative}
\end{cases}
\]

and:

\[
\text{level}_{\text{base}} = \text{baseMipLevel}
\]
\[ q = \text{levelCount} - 1 \]

`baseMipLevel` and `levelCount` are taken from the `subresourceRange` of the image view.

If the sampler’s `mipmapMode` is `VK_SAMPLER_MIPMAP_MODE_NEAREST`, then the level selected is \( d = d_l \).

If the sampler’s `mipmapMode` is `VK_SAMPLER_MIPMAP_MODE_LINEAR`, two neighboring levels are selected:

\[
\begin{align*}
&d_{hi} = |d_l| \\
&d_{lo} = \min(d_{hi} + 1, q) \\
&\delta = d_l - d_{hi}
\end{align*}
\]

\( \delta \) is the fractional value, quantized to the number of mipmap precision bits, used for linear filtering between levels.

### 16.5.8. (s,t,r,q,a) to (u,v,w,a) Transformation

The normalized texel coordinates are scaled by the image level dimensions and the array layer is selected.

This transformation is performed once for each level used in filtering (either \( d \), or \( d_{hi} \) and \( d_{lo} \)).

\[
\begin{align*}
&u(x, y) = s(x, y) \times width_{scale} + \Delta_i \\
&v(x, y) = \begin{cases} 
0 & \text{for 1D images} \\
\frac{t(x, y)}{width_{scale}} \times height_{scale} + \Delta_j & \text{otherwise}
\end{cases} \\
&w(x, y) = \begin{cases} 
0 & \text{for 2D or Cube images} \\
\frac{r(x, y)}{height_{scale}} \times depth_{scale} + \Delta_k & \text{otherwise}
\end{cases} \\
&a(x, y) = \begin{cases} 
\frac{a(x, y)}{depth_{scale}} & \text{for array images} \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]

where:

\[ width_{scale} = width_{level} \]

\[ height_{scale} = height_{level} \]

\[ depth_{scale} = depth_{level} \]

for conventional images, and:

\[ width_{scale} = width_{level} - 1 \]

\[ height_{scale} = height_{level} - 1 \]

---

Chapter 16. Image Operations | 1101
depth\_scale = depth\_level - 1

for corner-sampled images.

and where \((\Delta_u, \Delta_v, \Delta_w)\) are taken from the image instruction if it includes a \texttt{ConstOffset} or \texttt{Offset} operand, otherwise they are taken to be zero.

Operations then proceed to Unnormalized Texel Coordinate Operations.

### 16.6. Unnormalized Texel Coordinate Operations

#### 16.6.1. \((u,v,w,a)\) to \((i,j,k,l,n)\) Transformation And Array Layer Selection

The unnormalized texel coordinates are transformed to integer texel coordinates relative to the selected mipmap level.

The layer index \(l\) is computed as:

\[
l = \text{clamp}(\text{RNE}(a), 0, \text{layerCount} - 1) + \text{baseArrayLayer}
\]

where \text{layerCount} is the number of layers in the image subresource range of the image view, \text{baseArrayLayer} is the first layer from the subresource range, and where:

\[
\text{RNE}(a) = \begin{cases} \text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\ \lfloor a + 0.5 \rfloor & \text{alternative} \end{cases}
\]

The sample index \(n\) is assigned the value 0.

Nearest filtering (\texttt{VK\_FILTER\_NEAREST}) computes the integer texel coordinates that the unnormalized coordinates lie within:

\[
\begin{align*}
i &= \lfloor u + \text{shift} \rfloor \\
j &= \lfloor v + \text{shift} \rfloor \\
k &= \lfloor w + \text{shift} \rfloor
\end{align*}
\]

where:

\[
\text{shift} = 0.0
\]

for conventional images, and:

\[
\text{shift} = 0.5
\]

for corner-sampled images.

Linear filtering (\texttt{VK\_FILTER\_LINEAR}) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of \(i_0\) or \(i_1\), \(j_0\) or \(j_1\), \(k_0\) or \(k_1\),
as well as weights $\alpha$, $\beta$, and $\gamma$.

\[
\begin{align*}
i_0 &= \lfloor u - \text{shift} \rfloor \\
i_1 &= i_0 + 1 \\
j_0 &= \lfloor v - \text{shift} \rfloor \\
j_1 &= j_0 + 1 \\
k_0 &= \lfloor w - \text{shift} \rfloor \\
k_1 &= k_0 + 1
\end{align*}
\]

\[
\alpha = \text{frac}(u - \text{shift}) \\
\beta = \text{frac}(v - \text{shift}) \\
\gamma = \text{frac}(w - \text{shift})
\]

where:

\[
\text{shift} = 0.5
\]

for conventional images, and:

\[
\text{shift} = 0.0
\]

for corner-sampled images, and where:

\[
\text{frac}(x) = x - \lfloor x \rfloor
\]

where the number of fraction bits retained is specified by `VkPhysicalDeviceLimits::subTexelPrecisionBits`.

Cubic filtering (`VK_FILTER_CUBIC_EXT`) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of $i_0$, $i_1$, $i_2$, or $i_3$, $j_0$, $j_1$, $j_2$, or $j_3$, $k_0$, $k_1$, $k_2$, or $k_3$, as well as weights $\alpha$, $\beta$, and $\gamma$.

\[
\begin{align*}
i_0 &= \lfloor u - \frac{3}{2} \rfloor \\
i_1 &= i_0 + 1 \\
i_2 &= i_1 + 1 \\
i_3 &= i_2 + 1 \\
j_0 &= \lfloor v - \frac{3}{2} \rfloor \\
j_1 &= j_0 + 1 \\
j_2 &= j_1 + 1 \\
j_3 &= j_2 + 1 \\
k_0 &= \lfloor w - \frac{3}{2} \rfloor \\
k_1 &= k_0 + 1 \\
k_2 &= k_1 + 1 \\
k_3 &= k_2 + 1
\end{align*}
\]

\[
\alpha = \text{frac}(u - \frac{1}{2}) \\
\beta = \text{frac}(v - \frac{1}{2}) \\
\gamma = \text{frac}(w - \frac{1}{2})
\]

where:

\[
\text{frac}(x) = x - \lfloor x \rfloor
\]

where the number of fraction bits retained is specified by `VkPhysicalDeviceLimits`
::subTexelPrecisionBits.

16.7. Integer Texel Coordinate Operations

Integer texel coordinate operations may supply a LOD which texels are to be read from or written to using the optional SPIR-V operand Lod. If the Lod is provided then it must be an integer.

The image level selected is:

\[
d = \text{level}_{\text{base}} + \begin{cases} \text{Lod} & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases}
\]

If d does not lie in the range \([\text{baseMipLevel}, \text{baseMipLevel} + \text{levelCount})\) then any values fetched are zero if robustImageAccess2 is enabled, otherwise are undefined, and any writes (if supported) are discarded.

16.8. Image Sample Operations

16.8.1. Wrapping Operation

Cube images ignore the wrap modes specified in the sampler. Instead, if VK_FILTER_NEAREST is used within a mip level then VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE is used, and if VK_FILTER_LINEAR is used within a mip level then sampling at the edges is performed as described earlier in the Cube map edge handling section.

The first integer texel coordinate \(i\) is transformed based on the addressModeU parameter of the sampler.

\[
i = \begin{cases} i \mod \text{size} & \text{for repeat} \\ \text{clamp} (i, 0, \text{size}) & \text{for clamp to edge} \\ \text{clamp} (i - 1, \text{size}) \quad \text{for clamp to border} \\ \text{mirror} (i, 0, \text{size} - 1) \quad \text{for mirror clamp to edge} \end{cases}
\]

where:

\[
\text{mirror} (n) = \begin{cases} n & \text{for } n \geq 0 \\ -(1 + n) & \text{otherwise} \end{cases}
\]

\(j\) (for 2D and Cube image) and \(k\) (for 3D image) are similarly transformed based on the addressModeV and addressModeW parameters of the sampler, respectively.

16.8.2. Texel Gathering

SPIR-V instructions with Gather in the name return a vector derived from 4 texels in the base level of the image view. The rules for the VK_FILTER_LINEAR minification filter are applied to identify the four selected texels. Each texel is then converted to an RGBA value according to conversion to RGBA and then swizzled. A four-component vector is then assembled by taking the component indicated by the Component value in the instruction from the swizzled color value of the four texels. If the
If the operation does not use the `ConstOffsets` image operand then the four texels form the $2 \times 2$ rectangle used for texture filtering:

\[
\begin{align*}
\tau[R] &= \tau_{i,j,l}[\text{level}_\text{base}][\text{comp}] \\
\tau[G] &= \tau_{i,j,l}[\text{level}_\text{base}][\text{comp}] \\
\tau[B] &= \tau_{i,j,l}[\text{level}_\text{base}][\text{comp}] \\
\tau[A] &= \tau_{i,j,l}[\text{level}_\text{base}][\text{comp}]
\end{align*}
\]

If the operation does use the `ConstOffsets` image operand then the offsets allow a custom filter to be defined:

\[
\begin{align*}
\tau[R] &= \tau_{i,j,l} + \Delta_l[\text{level}_\text{base}][\text{comp}] \\
\tau[G] &= \tau_{i,j,l} + \Delta_l[\text{level}_\text{base}][\text{comp}] \\
\tau[B] &= \tau_{i,j,l} + \Delta_l[\text{level}_\text{base}][\text{comp}] \\
\tau[A] &= \tau_{i,j,l} + \Delta_l[\text{level}_\text{base}][\text{comp}]
\end{align*}
\]

where:

\[
\tau[\text{level}_\text{base}][\text{comp}] = \begin{cases} 
\tau[\text{level}_\text{base}][R], & \text{for } \text{comp} = 0 \\
\tau[\text{level}_\text{base}][G], & \text{for } \text{comp} = 1 \\
\tau[\text{level}_\text{base}][B], & \text{for } \text{comp} = 2 \\
\tau[\text{level}_\text{base}][A], & \text{for } \text{comp} = 3 
\end{cases}
\]

`OpImage*Gather` **must** not be used on a sampled image with `sampler YCBCR` conversion enabled.

### 16.8.3. Texel Filtering

Texel filtering is first performed for each level (either $d$ or $d_{hi}$ and $d_{lo}$).

If $\lambda$ is less than or equal to zero, the texture is said to be **magnified**, and the filter mode within a mip level is selected by the `magFilter` in the sampler. If $\lambda$ is greater than zero, the texture is said to be **minified**, and the filter mode within a mip level is selected by the `minFilter` in the sampler.

#### Texel Nearest Filtering

Within a mip level, `VK_FILTER_NEAREST` filtering selects a single value using the $(i, j, k)$ texel coordinates, with all texels taken from layer $l$.

\[
\tau[\text{level}] = \begin{cases} 
\tau_{i,k}[\text{level}], & \text{for 3D image} \\
\tau_{i,j}[\text{level}], & \text{for 2D or Cube image} \\
\tau_{i,j}[\text{level}], & \text{for 1D image}
\end{cases}
\]

#### Texel Linear Filtering

Within a mip level, `VK_FILTER_LINEAR` filtering combines 8 (for 3D), 4 (for 2D or Cube), or 2 (for 1D) texel values, together with their linear weights. The linear weights are derived from the fractions computed earlier:
The values of multiple texels, together with their weights, are combined to produce a filtered value.

The `VkSamplerReductionModeCreateInfo::reductionMode` can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the `reductionMode` is set (explicitly or implicitly) to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`, a weighted average is computed:

\[
\tau_{3D} = \sum_{k=k_0}^{k_1} \sum_{j=j_0}^{j_1} \sum_{i=i_0}^{i_1} (w_i)(w_j)(w_k)\tau_{ijk} \\
\tau_{2D} = \sum_{j=j_0}^{j_1} \sum_{i=i_0}^{i_1} (w_i)(w_j)\tau_{ij} \\
\tau_{1D} = \sum_{i=i_0}^{i_1} (w_i)\tau_i
\]

However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX`, the process operates on the above set of multiple texels, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the set of texels with non-zero weights.

**Texel Cubic Filtering**

Within a mip level, `VK_FILTER_CUBIC_EXT`, filtering computes a weighted average of 64 (for 3D), 16 (for 2D), or 4 (for 1D) texel values, together with their Catmull-Rom weights.

Catmull-Rom weights are derived from the fractions computed earlier.

\[
\begin{bmatrix}
w_{i_0} & w_{i_1} & w_{i_2} & w_{i_3}
\end{bmatrix} = \frac{1}{24}\begin{bmatrix}1 & \alpha & \alpha^2 & \alpha^3
\end{bmatrix}
\]

\[
\begin{bmatrix}
w_{j_0} & w_{j_1} & w_{j_2} & w_{j_3}
\end{bmatrix} = \frac{1}{24}\begin{bmatrix}1 & \beta & \beta^2 & \beta^3
\end{bmatrix}
\]

\[
\begin{bmatrix}
w_{k_0} & w_{k_1} & w_{k_2} & w_{k_3}
\end{bmatrix} = \frac{1}{24}\begin{bmatrix}1 & \gamma & \gamma^2 & \gamma^3
\end{bmatrix}
\]

The values of multiple texels, together with their weights, are combined to produce a filtered value.

The `VkSamplerReductionModeCreateInfo::reductionMode` can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the `reductionMode` is set (explicitly or implicitly) to
VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE, a weighted average is computed:

\[
\tau_{3D} = \sum_{k=0}^{k_3} \sum_{j=0}^{j_3} \sum_{i=0}^{i_3} (w_j)(w_k)\tau_{ijk}
\]

\[
\tau_{2D} = \sum_{j=0}^{j_3} \sum_{i=0}^{i_3} (w_j)\tau_{ij}
\]

\[
\tau_{1D} = \sum_{i=0}^{i_3} (w_i)\tau_i
\]

However, if the reduction mode is VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX, the process operates on the above set of multiple texels, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the set of texels with non-zero weights.

Texel Mipmap Filtering

VK_SAMPLER_MIPMAP_MODE_NEAREST filtering returns the value of a single mipmap level,

\[\tau = \tau[d].\]

VK_SAMPLER_MIPMAP_MODE_LINEAR filtering combines the values of multiple mipmap levels (\(\tau[hi]\) and \(\tau[lo]\)), together with their linear weights.

The linear weights are derived from the fraction computed earlier:

\[w_{hi} = (1 - \delta)\]
\[w_{lo} = (\delta)\]

The values of multiple mipmap levels, together with their weights, are combined to produce a final filtered value.

The VkSamplerReductionModeCreateInfo::reductionMode can control the process by which multiple texels, together with their weights, are combined to produce a filtered texture value.

When the reductionMode is set (explicitly or implicitly) to VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE, a weighted average is computed:

\[\tau = (w_{hi})\tau[hi] + (w_{lo})\tau[lo]\]

Texel Anisotropic Filtering

Anisotropic filtering is enabled by the anisotropyEnable in the sampler. When enabled, the image filtering scheme accounts for a degree of anisotropy.

The particular scheme for anisotropic texture filtering is implementation dependent. Implementations should consider the magFilter, minFilter and.mipmapMode of the sampler to control the specifics of the anisotropic filtering scheme used. In addition, implementations should consider minLod and maxLod of the sampler.

The following describes one particular approach to implementing anisotropic filtering for the 2D Image case, implementations may choose other methods:
Given a `magFilter`, `minFilter` of `VK_FILTER_LINEAR` and a `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_NEAREST`:

Instead of a single isotropic sample, N isotropic samples are sampled within the image footprint of the image level d to approximate an anisotropic filter. The sum $\tau_{2D_{\text{aniso}}}$ is defined using the single isotropic $\tau_{2D}(u,v)$ at level d.

$$
\tau_{2D_{\text{aniso}}} = \frac{1}{N} \sum_{i=0}^{N-1} \tau_{2D}(u - \frac{1}{2} \frac{i}{N+1}, v - \frac{1}{2} \frac{i}{N+1}), \quad \text{when } \rho_x > \rho_y
$$

$$
\tau_{2D_{\text{aniso}}} = \frac{1}{N} \sum_{i=0}^{N-1} \tau_{2D}(u, v - \frac{1}{2} \frac{i}{N+1}), \quad \text{when } \rho_y \geq \rho_x
$$

When `VkSamplerReductionModeCreateInfo::reductionMode` is set to `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE`, the above summation is used. However, if the reduction mode is `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX`, the process operates on the above values, together with their weights, computing a component-wise minimum or maximum, respectively, of the components of the values with non-zero weights.

### 16.9. Texel Footprint Evaluation

The SPIR-V instruction `OpImageSampleFootprintNV` evaluates the set of texels from a single mip level that would be accessed during a texel filtering operation. In addition to the inputs that would be accepted by an equivalent `OpImageSample*` instruction, `OpImageSampleFootprintNV` accepts two additional inputs. The `Granularity` input is an integer identifying the size of texel groups used to evaluate the footprint. Each bit in the returned footprint mask corresponds to an aligned block of texels whose size is given by the following table:

Table 29. Texel footprint granularity values

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Dim = 2D</th>
<th>Dim = 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unsupported</td>
<td>unsupported</td>
</tr>
<tr>
<td>1</td>
<td>2x2</td>
<td>2x2x2</td>
</tr>
<tr>
<td>2</td>
<td>4x2</td>
<td>unsupported</td>
</tr>
<tr>
<td>3</td>
<td>4x4</td>
<td>4x4x2</td>
</tr>
<tr>
<td>4</td>
<td>8x4</td>
<td>unsupported</td>
</tr>
<tr>
<td>5</td>
<td>8x8</td>
<td>unsupported</td>
</tr>
<tr>
<td>6</td>
<td>16x8</td>
<td>unsupported</td>
</tr>
<tr>
<td>7</td>
<td>16x16</td>
<td>unsupported</td>
</tr>
<tr>
<td>8</td>
<td>unsupported</td>
<td>unsupported</td>
</tr>
<tr>
<td>9</td>
<td>unsupported</td>
<td>unsupported</td>
</tr>
<tr>
<td>10</td>
<td>unsupported</td>
<td>16x16x16</td>
</tr>
<tr>
<td>11</td>
<td>64x64</td>
<td>32x16x16</td>
</tr>
<tr>
<td>12</td>
<td>128x64</td>
<td>32x32x16</td>
</tr>
</tbody>
</table>
The **Coarse** input is used to select between the two mip levels that may be accessed during texel filtering when using a `mipmapMode` of `VK_SAMPLER_MIPMAP_MODE_LINEAR`. When filtering between two mip levels, a `Coarse` value of `true` requests the footprint in the lower-resolution mip level (higher level number), while `false` requests the footprint in the higher-resolution mip level. If texel filtering would access only a single mip level, the footprint in that level would be returned when `Coarse` is set to `false`; an empty footprint would be returned when `Coarse` is set to `true`.

The footprint for `OpImageSampleFootprintNV` is returned in a structure with six members:

- The first member is a boolean value that is true if the texel filtering operation would access only a single mip level.
- The second member is a two- or three-component integer vector holding the footprint anchor location. For two-dimensional images, the returned components are in units of eight texel groups. For three-dimensional images, the returned components are in units of four texel groups.
- The third member is a two- or three-component integer vector holding a footprint offset relative to the anchor. All returned components are in units of texel groups.
- The fourth member is a two-component integer vector mask, which holds a bitfield identifying the set of texel groups in an 8x8 or 4x4x4 neighborhood relative to the anchor and offset.
- The fifth member is an integer identifying the mip level containing the footprint identified by the anchor, offset, and mask.
- The sixth member is an integer identifying the granularity of the returned footprint.

For footprints in two-dimensional images (Dim2D), the mask returned by `OpImageSampleFootprintNV` indicates whether each texel group in a 8x8 local neighborhood of texel groups would have one or more texels accessed during texel filtering. In the mask, the texel group with local group coordinates \((lgx, lgy)\) is considered covered if and only if

\[
0 \neq (\text{mask}.x + (\text{mask}.y\ll 32))\&(1\ll(lgy \times 8 + lgx))
\]

where:

- \(0 \leq lgx<8\) and \(0\leq lgy<8\); and
- `mask` is the returned two-component mask.

The local group with coordinates \((lgx, lgy)\) in the mask is considered covered if and only if the texel filtering operation would access one or more texels \(t_{ij}\) in the returned miplevel where:
\( m0 = \begin{cases} \text{gran.x} \times (8 \times \text{anchor.x} + lgx), & \text{if} lgx + \text{offset}.x < 8 \\ \text{gran.x} \times (8 \times (\text{anchor.x} - 1) + lgx), & \text{otherwise} \end{cases} \)
\( m1 = m0 + \text{gran.x} - 1 \)
\( m0 = \begin{cases} \text{gran.y} \times (8 \times \text{anchor.y} + lgy), & \text{if} lgy + \text{offset}.y < 8 \\ \text{gran.y} \times (8 \times (\text{anchor.y} - 1) + lgy), & \text{otherwise} \end{cases} \)
\( m1 = m0 + \text{gran.y} - 1 \)

and

- \( i0 < i <= i1 \) and \( j0 < j <= j1 \);
- \textit{gran} is a two-component vector holding the width and height of the texel group identified by the granularity;
- \textit{anchor} is the returned two-component anchor vector; and
- \textit{offset} is the returned two-component offset vector.

For footprints in three-dimensional images (Dim3D), the mask returned by \texttt{OpImageSampleFootprintNV} indicates whether each texel group in a 4x4x4 local neighborhood of texel groups would have one or more texels accessed during texel filtering. In the mask, the texel group with local group coordinates \((lgx, lgy, lgz)\), is considered covered if and only if:

\[
0 \neq ((\text{mask}.x + (\text{mask}.y<<32))&,(1<<(lgz \times 16+lgy \times 4+lgx)))
\]

where:

- \( 0 < lgx < 4, 0 < lgy < 4, \) and \( 0 < lgz < 4; \) and
- \textit{mask} is the returned two-component mask.

The local group with coordinates \((lgx, lgy, lgz)\) in the mask is considered covered if and only if the texel filtering operation would access one or more texels \(t_{i,j,k}\) in the returned mipmap level where:

\( m0 = \begin{cases} \text{gran}.x \times (4 \times \text{anchor}.x + lgx), & \text{if} lgx + \text{offset}.x < 4 \\ \text{gran}.x \times (4 \times (\text{anchor}.x - 1) + lgx), & \text{otherwise} \end{cases} \)
\( m1 = m0 + \text{gran}.x - 1 \)
\( m0 = \begin{cases} \text{gran}.y \times (4 \times \text{anchor}.y + lgy), & \text{if} lgy + \text{offset}.y < 4 \\ \text{gran}.y \times (4 \times (\text{anchor}.y - 1) + lgy), & \text{otherwise} \end{cases} \)
\( m1 = m0 + \text{gran}.y - 1 \)
\( m0 = \begin{cases} \text{gran}.z \times (4 \times \text{anchor}.z + lgz), & \text{if} lgz + \text{offset}.z < 4 \\ \text{gran}.z \times (4 \times (\text{anchor}.z - 1) + lgz), & \text{otherwise} \end{cases} \)
\( m1 = m0 + \text{gran}.z - 1 \)

and

- \( i0 < i <= i1, j0 < j <= j1, k0 < k <= k1; \)
- \textit{gran} is a three-component vector holding the width, height, and depth of the texel group identified by the granularity;
- \textit{anchor} is the returned three-component anchor vector; and
- \textit{offset} is the returned three-component offset vector.

If the sampler used by \texttt{OpImageSampleFootprintNV} enables anisotropic texel filtering via
anisotropyEnable, it is possible that the set of texel groups accessed in a mip level may be too large to be expressed using an 8x8 or 4x4x4 mask using the granularity requested in the instruction. In this case, the implementation uses a texel group larger than the requested granularity. When a larger texel group size is used, OpImageSampleFootprintNV returns an integer granularity value that can be interpreted in the same manner as the granularity value provided to the instruction to determine the texel group size used. If anisotropic texel filtering is disabled in the sampler, or if an anisotropic footprint can be represented as an 8x8 or 4x4x4 mask with the requested granularity, OpImageSampleFootprintNV will use the requested granularity as-is and return a granularity value of zero.

OpImageSampleFootprintNV supports only two- and three-dimensional image accesses (Dim2D and Dim3D), and the footprint returned is undefined if a sampler uses an addressing mode other than VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE.

16.10. Image Operation Steps

Each step described in this chapter is performed by a subset of the image instructions:

- Texel Input Validation Operations, Format Conversion, Texel Replacement, Conversion to RGBA, and Component Swizzle: Performed by all instructions except OpImageWrite.
- Depth Comparison: Performed by OpImage*Dref instructions.
- All Texel output operations: Performed by OpImageWrite.
- Projection: Performed by all OpImage*Proj instructions.
- Derivative Image Operations, Cube Map Operations, Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection, and Texel Anisotropic Filtering: Performed by all OpImageSample* and OpImageSparseSample* instructions.
- (s,t,r,q,a) to (u,v,w,a) Transformation, Wrapping, and (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection: Performed by all OpImageSample, OpImageSparseSample, and OpImage*Gather instructions.
- Texel Filtering: Performed by all OpImageSample* and OpImageSparseSample* instructions.
- Sparse Residency: Performed by all OpImageSparse* instructions.

16.11. Image Query Instructions

16.11.1. Image Property Queries

OpImageQuerySize, OpImageQuerySizeLod, OpImageQueryLevels, and OpImageQuerySamples query properties of the image descriptor that would be accessed by a shader image operation. They return 0 if the bound descriptor is a null descriptor.

OpImageQuerySizeLod returns the size of the image level identified by the Level of Detail operand. If that level does not exist in the image, and the descriptor is not null, then the value returned is
16.11.2. Lod Query

OpImageQueryLod returns the Lod parameters that would be used in an image operation with the given image and coordinates. If the descriptor that would be accessed is a null descriptor then (0, 0) is returned. Otherwise, the steps described in this chapter are performed as if for OpImageSampleImplicitLod, up to Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection. The return value is the vector \((\lambda', d)\). These values may be subject to implementation-specific maxima and minima for very large, out-of-range values.
Chapter 17. Fragment Density Map Operations

17.1. Fragment Density Map Operations Overview

When a fragment is generated in a render pass that has a fragment density map attachment, its area is determined by the properties of the local framebuffer region that the fragment occupies. The framebuffer is divided into a uniform grid of these local regions, and their fragment area property is derived from the density map with the following operations:

- Fetch density value
  - Component swizzle
  - Component mapping
- Fragment area conversion
  - Fragment area filter
  - Fragment area clamp

17.2. Fetch Density Value

Each local framebuffer region at center coordinate \((x, y)\) fetches a texel from the fragment density map at integer coordinates:

\[
i = \left\lfloor \frac{x}{\text{Fragment Density Texel Size}_\text{width}} \right\rfloor
\]

\[
j = \left\lfloor \frac{y}{\text{Fragment Density Texel Size}_\text{height}} \right\rfloor
\]

Where the size of each region in the framebuffer is:

\[
\text{fragment Density Texel Size}_\text{width} = 2^{\left\lfloor \log_2 \left( \frac{\text{Framebuffer Width}}{\text{Fragment Density Map Width}} \right) \right\rfloor}
\]

\[
\text{fragment Density Texel Size}_\text{height} = 2^{\left\lfloor \log_2 \left( \frac{\text{Framebuffer Height}}{\text{Fragment Density Map Height}} \right) \right\rfloor}
\]

This region is subject to the limits in \text{VkPhysicalDeviceFragmentDensityMapPropertiesEXT} and therefore the final region size is clamped:

\[
\text{fragment Density Texel Size}_{\text{width}} = \text{clamp} \left( \text{fragment Density Texel Size}_\text{width}, \min \text{Fragment Density Texel Size}_{\text{width}}, \max \text{Fragment Density Texel Size}_{\text{width}} \right)
\]

\[
\text{fragment Density Texel Size}_{\text{height}} = \text{clamp} \left( \text{fragment Density Texel Size}_\text{height}, \min \text{Fragment Density Texel Size}_{\text{height}}, \max \text{Fragment Density Texel Size}_{\text{height}} \right)
\]
When multiview is enabled for the render pass and the fragment density map attachment view was created with `layerCount` greater than 1, the density map layer that the texel is fetched from is:

\[
layer = baseArrayLayer + ViewIndex
\]

Otherwise:

\[
layer = baseArrayLayer
\]

The texel fetched from the density map at (i,j,layer) is next converted to density with the following operations.

### 17.2.1. Component Swizzle

The `components` member of `VkImageViewCreateInfo` is applied to the fetched texel as defined in [Image component swizzle](#).

### 17.2.2. Component Mapping

The swizzled texel's components are mapped to a density value:

\[
densityValue_{xy} = (C^r, C^g)
\]

### 17.3. Fragment Area Conversion

Fragment area for the framebuffer region is undefined if the density fetched is not a normalized floating-point value greater than 0.0. Otherwise, the fetched fragment area for that region is derived as:

\[
fragmentArea_{wh} = \frac{1.0}{densityValue_{xy}}
\]

### 17.3.1. Fragment Area Filter

Optionally, the implementation may fetch additional density map texels in an implementation defined window around (i,j). The texels follow the standard conversion steps up to and including fragment area conversion.

A single fetched fragment area for the framebuffer region is chosen by the implementation and must have an area between the `min` and `max` areas of the fetched set.
17.3.2. Fragment Area Clamp

The implementation may clamp the fetched fragment area to one that it supports. The clamped fragment area must have a size less than or equal to the original fetched value. Implementations may vary the supported set of fragment areas per framebuffer region. Fragment area (1,1) must always be in the supported set.

Note

For example, if the fetched fragment area is (1,4) but the implementation only supports areas of {{1,1},{2,2}}, it could choose to clamp the area to (2,2) since it has the same size as (1,4). While this would produce fragments that have lower quality strictly in the x-axis, the overall density is maintained.

The clamped fragment area is assigned to the corresponding framebuffer region.
Chapter 18. Queries

Queries provide a mechanism to return information about the processing of a sequence of Vulkan commands. Query operations are asynchronous, and as such, their results are not returned immediately. Instead, their results, and their availability status are stored in a Query Pool. The state of these queries can be read back on the host, or copied to a buffer object on the device.

The supported query types are Occlusion Queries, Pipeline Statistics Queries, Result Status Queries, Video Encode Bitstream Queries, and Timestamp Queries. Performance Queries are supported if the associated extension is available. Transform Feedback Queries are supported if the associated extension is available. Intel performance queries are supported if the associated extension is available.

Several additional queries with specific purposes associated with ray tracing are available if the corresponding extensions are supported, as described for VkQueryType.

18.1. Query Pools

Queries are managed using query pool objects. Each query pool is a collection of a specific number of queries of a particular type.

Query pools are represented by VkQueryPool handles:

```c
// Provided by VK_VERSION_1_0
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkQueryPool)
```

To create a query pool, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkCreateQueryPool(
    VkDevice device,
    const VkQueryPoolCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkQueryPool* pQueryPool);
```

- `device` is the logical device that creates the query pool.
- `pCreateInfo` is a pointer to a VkQueryPoolCreateInfo structure containing the number and type of queries to be managed by the pool.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pQueryPool` is a pointer to a VkQueryPool handle in which the resulting query pool object is returned.
Valid Usage (Implicit)

- VUID-vkCreateQueryPool-device-parameter
  
  *device* must be a valid *VkDevice* handle

- VUID-vkCreateQueryPool-pCreateInfo-parameter
  
  *pNext* must be a valid pointer to a valid *VkQueryPoolCreateInfo* structure

- VUID-vkCreateQueryPool-pAllocator-parameter
  
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- VUID-vkCreateQueryPool-pQueryPool-parameter
  
  *pQueryPool* must be a valid pointer to a *VkQueryPool* handle

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The *VkQueryPoolCreateInfo* structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkQueryPoolCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkQueryPoolCreateFlags flags;
    VkQueryType queryType;
    uint32_t queryCount;
    VkQueryPipelineStatisticFlags pipelineStatistics;
} VkQueryPoolCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *flags* is reserved for future use.
- *queryType* is a *VkQueryType* value specifying the type of queries managed by the pool.
- *queryCount* is the number of queries managed by the pool.
- *pipelineStatistics* is a bitmask of *VkQueryPipelineStatisticFlagBits* specifying which counters will be returned in queries on the new pool, as described below in Pipeline Statistics Queries.

*pipelineStatistics* is ignored if *queryType* is not VK_QUERY_TYPE_PIPELINE_STATISTICS.
Valid Usage

- **VUID-VkQueryPoolCreateInfo-queryType-00791**
  
  If the pipeline statistics queries feature is not enabled, `queryType` must not be `VK_QUERY_TYPE_PIPELINE_STATISTICS`.

- **VUID-VkQueryPoolCreateInfo-queryType-00792**
  
  If `queryType` is `VK_QUERY_TYPE_PIPELINE_STATISTICS`, `pipelineStatistics` must be a valid combination of `VkQueryPipelineStatisticFlagBits` values.

- **VUID-VkQueryPoolCreateInfo-queryType-03222**
  
  If `queryType` is `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `pNext` chain must include a structure of type `VkQueryPoolPerformanceCreateInfoKHR`.

- **VUID-VkQueryPoolCreateInfo-queryCount-02763**
  
  `queryCount` must be greater than 0.

Valid Usage (Implicit)

- **VUID-VkQueryPoolCreateInfo-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO`.

- **VUID-VkQueryPoolCreateInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkQueryPoolPerformanceCreateInfoKHR`, `VkQueryPoolPerformanceQueryCreateInfoINTEL`, or `VkVideoProfileKHR`.

- **VUID-VkQueryPoolCreateInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkQueryPoolCreateInfo-flags-zerobitmask**
  
  `flags` must be `0`.

- **VUID-VkQueryPoolCreateInfo-queryType-parameter**
  
  `queryType` must be a valid `VkQueryType` value.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryPoolCreateFlags;
```

`VkQueryPoolCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

The `VkQueryPoolPerformanceCreateInfoKHR` structure is defined as:
// Provided by VK_KHR_performance_query

typedef struct VkQueryPoolPerformanceCreateInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           queueFamilyIndex;
    uint32_t           counterIndexCount;
    const uint32_t*    pCounterIndices;
} VkQueryPoolPerformanceCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **queueFamilyIndex** is the queue family index to create this performance query pool for.
- **counterIndexCount** is size of the **pCounterIndices** array.
- **pCounterIndices** is the array of indices into the `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR::pCounters` to enable in this performance query pool.

**Valid Usage**

- VUID-VkQueryPoolPerformanceCreateInfoKHR-queueFamilyIndex-03236
  `queueFamilyIndex` **must** be a valid queue family index of the device
- VUID-VkQueryPoolPerformanceCreateInfoKHR-performanceCounterQueryPools-03237
  The `performanceCounterQueryPools` feature **must** be enabled
- VUID-VkQueryPoolPerformanceCreateInfoKHR-pCounterIndices-03321
  Each element of **pCounterIndices** **must** be in the range of counters reported by `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR` for the queue family specified in `queueFamilyIndex`

**Valid Usage (Implicit)**

- VUID-VkQueryPoolPerformanceCreateInfoKHR-sType-sType
  **sType** **must** be `VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR`
- VUID-VkQueryPoolPerformanceCreateInfoKHR-pCounterIndices-parameter
  **pCounterIndices** **must** be a valid pointer to an array of `counterIndexCount uint32_t` values
- VUID-VkQueryPoolPerformanceCreateInfoKHR-counterIndexCount-arraylength
  **counterIndexCount** **must** be greater than 0

To query the number of passes required to query a performance query pool on a physical device, call:
```c
// Provided by VK_KHR_performance_query
void vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR(
    VkPhysicalDevice physicalDevice,
    const VkQueryPoolPerformanceCreateInfoKHR* pPerformanceQueryCreateInfo,
    uint32_t* pNumPasses);
```

- `physicalDevice` is the handle to the physical device whose queue family performance query counter properties will be queried.
- `pPerformanceQueryCreateInfo` is a pointer to a `VkQueryPoolPerformanceCreateInfoKHR` of the performance query that is to be created.
- `pNumPasses` is a pointer to an integer related to the number of passes required to query the performance query pool, as described below.

The `pPerformanceQueryCreateInfo` member `VkQueryPoolPerformanceCreateInfoKHR::queueFamilyIndex` must be a queue family of `physicalDevice`. The number of passes required to capture the counters specified in the `pPerformanceQueryCreateInfo` member `VkQueryPoolPerformanceCreateInfoKHR :: pCounters` is returned in `pNumPasses`.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR-pPerformanceQueryCreateInfo-parameter
  `pPerformanceQueryCreateInfo` must be a valid pointer to a valid `VkQueryPoolPerformanceCreateInfoKHR` structure
- VUID-vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR-pNumPasses-parameter
  `pNumPasses` must be a valid pointer to a `uint32_t` value

To destroy a query pool, call:

```c
// Provided by VK_VERSION_1_0
void vkDestroyQueryPool(
    VkDevice device,
    VkQueryPool queryPool,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the query pool.
- `queryPool` is the query pool to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
Valid Usage

- VUID-vkDestroyQueryPool-queryPool-00793
  All submitted commands that refer to `queryPool` must have completed execution

- VUID-vkDestroyQueryPool-queryPool-00794
  If `VkAllocationCallbacks` were provided when `queryPool` was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyQueryPool-queryPool-00795
  If no `VkAllocationCallbacks` were provided when `queryPool` was created, `pAllocator` must be `NULL`

Valid Usage (Implicit)

- VUID-vkDestroyQueryPool-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkDestroyQueryPool-queryPool-parameter
  If `queryPool` is not `VK_NULL_HANDLE`, `queryPool` must be a valid `VkQueryPool` handle

- VUID-vkDestroyQueryPool-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkDestroyQueryPool-queryPool-parent
  If `queryPool` is a valid handle, it must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `queryPool` must be externally synchronized

Possible values of `VkQueryPoolCreateInfo::queryType`, specifying the type of queries managed by the pool, are:
typedef enum VkQueryType {
    VK_QUERY_TYPE_OCCLUSION = 0,
    VK_QUERY_TYPE_PIPELINE_STATISTICS = 1,
    VK_QUERY_TYPE_TIMESTAMP = 2,
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_queue
        VK_QUERY_TYPE_RESULT_STATUS_ONLY_KHR = 1000023000,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
        // Provided by VK_KHR_video_encode_queue
        VK_QUERY_TYPE_VIDEO_ENCODE_BITSTREAM_BUFFER_RANGE_KHR = 1000299000,
    #endif
    // Provided by VK_EXT_transform_feedback
    VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT = 1000028004,
    // Provided by VK_KHR_performance_query
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR = 1000116000,
    // Provided by VK_KHR_acceleration_structure
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR = 1000150000,
    // Provided by VK_KHR_acceleration_structure
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR = 1000150001,
    // Provided by VK_NV_ray_tracing
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV = 1000165000,
    // Provided by VK_INTEL_performance_query
    VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL = 1000210000,
    #endif
    VK_QUERY_TYPE_OCCLUSION
        specifies an occlusion query.
    VK_QUERY_TYPE_PIPELINE_STATISTICS
        specifies a pipeline statistics query.
    VK_QUERY_TYPE_TIMESTAMP
        specifies a timestamp query.
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR
        specifies a performance query.
    VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT
        specifies a transform feedback query.
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR
        specifies an acceleration structure size query for use with 
vkCmdWriteAccelerationStructuresPropertiesKHR
        or 
vkWriteAccelerationStructuresPropertiesKHR.
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR
        specifies a serialization acceleration structure size query
    VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV
        specifies an acceleration structure size query for use with 
vkCmdWriteAccelerationStructuresPropertiesNV.
    VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL
        specifies an Intel performance query.
    VK_QUERY_TYPE_RESULT_STATUS_ONLY_KHR
        specifies a result status query.
    VK_QUERY_TYPE_VIDEO_ENCODE_BITSTREAM_BUFFER_RANGE_KHR
        specifies a video encode bitstream range query.
} VkQueryType;
18.2. Query Operation

The operation of queries is controlled by the commands `vkCmdBeginQuery`, `vkCmdEndQuery`, `vkCmdBeginQueryIndexedEXT`, `vkCmdEndQueryIndexedEXT`, `vkCmdResetQueryPool`, `vkCmdCopyQueryPoolResults`, `vkCmdWriteTimestamp2KHR`, and `vkCmdWriteTimestamp`.

In order for a `VkCommandBuffer` to record query management commands, the queue family for which its `VkCommandPool` was created must support the appropriate type of operations (graphics, compute) suitable for the query type of a given query pool.

Each query in a query pool has a status that is either unavailable or available, and also has state to store the numerical results of a query operation of the type requested when the query pool was created. Resetting a query via `vkCmdResetQueryPool` or `vkResetQueryPool` sets the status to unavailable and makes the numerical results undefined. Performing a query operation with `vkCmdBeginQuery` and `vkCmdEndQuery` changes the status to available when the query finishes, and updates the numerical results. Both the availability status and numerical results are retrieved by calling either `vkGetQueryPoolResults` or `vkCmdCopyQueryPoolResults`.

Query commands, for the same query and submitted to the same queue, execute in their entirety in submission order, relative to each other. In effect there is an implicit execution dependency from each such query command to all query commands previously submitted to the same queue. There is one significant exception to this; if the `flags` parameter of `vkCmdCopyQueryPoolResults` does not include `VK_QUERY_RESULT_WAIT_BIT`, execution of `vkCmdCopyQueryPoolResults` may happen-before the results of `vkCmdEndQuery` are available.

After query pool creation, each query must be reset before it is used. Queries must also be reset between uses.

If a logical device includes multiple physical devices, then each command that writes a query must execute on a single physical device, and any call to `vkCmdBeginQuery` must execute the corresponding `vkCmdEndQuery` command on the same physical device.

To reset a range of queries in a query pool on a queue, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdResetQueryPool(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the handle of the query pool managing the queries being reset.
- `firstQuery` is the initial query index to reset.
- `queryCount` is the number of queries to reset.

When executed on a queue, this command sets the status of query indices `[firstQuery, firstQuery + queryCount)` to unavailable and makes the numerical results undefined.
queryCount - 1] to unavailable.

If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, this command sets the status of query indices [firstQuery, firstQuery + queryCount - 1] to unavailable for each pass of queryPool, as indicated by a call to vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR.

Note
Because vkCmdResetQueryPool resets all the passes of the indicated queries, applications must not record a vkCmdResetQueryPool command for a queryPool created with VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR in a command buffer that needs to be submitted multiple times as indicated by a call to vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR. Otherwise applications will never be able to complete the recorded queries.

Valid Usage

- VUID-vkCmdResetQueryPool-firstQuery-00796
  firstQuery must be less than the number of queries in queryPool

- VUID-vkCmdResetQueryPool-firstQuery-00797
  The sum of firstQuery and queryCount must be less than or equal to the number of queries in queryPool

- VUID-vkCmdResetQueryPool-None-02841
  All queries used by the command must not be active

- VUID-vkCmdResetQueryPool-firstQuery-02862
  If queryPool was created with VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, this command must not be recorded in a command buffer that, either directly or through secondary command buffers, also contains begin commands for a query from the set of queries [firstQuery, firstQuery + queryCount - 1]
**Valid Usage (Implicit)**

- VUID-vkCmdResetQueryPool-commandBuffer-parameter  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdResetQueryPool-queryPool-parameter  
  `queryPool` must be a valid `VkQueryPool` handle

- VUID-vkCmdResetQueryPool-commandBuffer-recording  
  `commandBuffer` must be in the recording state

- VUID-vkCmdResetQueryPool-commandBuffer-cmdpool  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- VUID-vkCmdResetQueryPool-renderpass  
  This command must only be called outside of a render pass instance

- VUID-vkCmdResetQueryPool-commonparent  
  Both of `commandBuffer`, and `queryPool` must have been created, allocated, or retrieved from the same `VkDevice`

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To reset a range of queries in a query pool on the host, call:

```c
// Provided by VK_EXT_host_query_reset
void vkResetQueryPoolEXT(
    VkDevice device,  
    VkQueryPool queryPool,  
    uint32_t firstQuery,  
    uint32_t queryCount);
```

- `device` is the logical device that owns the query pool.
• queryPool is the handle of the query pool managing the queries being reset.
• firstQuery is the initial query index to reset.
• queryCount is the number of queries to reset.

This command sets the status of query indices \([firstQuery, firstQuery + queryCount - 1]\) to unavailable.

If queryPool is VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR this command sets the status of query indices \([firstQuery, firstQuery + queryCount - 1]\) to unavailable for each pass.

---

**Valid Usage**

- VUID-vkResetQueryPool-None-02665
  The hostQueryReset feature **must** be enabled

- VUID-vkResetQueryPool-firstQuery-02666
  firstQuery **must** be less than the number of queries in queryPool

- VUID-vkResetQueryPool-firstQuery-02667
  The sum of firstQuery and queryCount **must** be less than or equal to the number of queries in queryPool

- VUID-vkResetQueryPool-firstQuery-02741
  Submitted commands that refer to the range specified by firstQuery and queryCount in queryPool **must** have completed execution

- VUID-vkResetQueryPool-firstQuery-02742
  The range of queries specified by firstQuery and queryCount in queryPool **must** not be in use by calls to `vkGetQueryPoolResults` or `vkResetQueryPool` in other threads

---

**Valid Usage (Implicit)**

- VUID-vkResetQueryPool-device-parameter
  device **must** be a valid VkDevice handle

- VUID-vkResetQueryPool-queryPool-parameter
  queryPool **must** be a valid VkQueryPool handle

- VUID-vkResetQueryPool-queryPool-parent
  queryPool **must** have been created, allocated, or retrieved from device

---

Once queries are reset and ready for use, query commands **can** be issued to a command buffer. Occlusion queries and pipeline statistics queries count events - drawn samples and pipeline stage invocations, respectively - resulting from commands that are recorded between a `vkCmdBeginQuery` command and a `vkCmdEndQuery` command within a specified command buffer, effectively scoping a set of drawing and/or dispatch commands. Timestamp queries write timestamps to a query pool. Performance queries record performance counters to a query pool.

A query **must** begin and end in the same command buffer, although if it is a primary command
buffer, and the inherited queries feature is enabled, it can execute secondary command buffers during the query operation. For a secondary command buffer to be executed while a query is active, it must set the occlusionQueryEnable, queryFlags, and/or pipelineStatistics members of VkCommandBufferInheritanceInfo to conservative values, as described in the Command Buffer Recording section. A query must either begin and end inside the same subpass of a render pass instance, or must both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

If queries are used while executing a render pass instance that has multiview enabled, the query uses N consecutive query indices in the query pool (starting at query) where N is the number of bits set in the view mask in the subpass the query is used in. How the numerical results of the query are distributed among the queries is implementation-dependent. For example, some implementations may write each view's results to a distinct query, while other implementations may write the total result to the first query and write zero to the other queries. However, the sum of the results in all the queries must accurately reflect the total result of the query summed over all views. Applications can sum the results from all the queries to compute the total result.

Queries used with multiview rendering must not span subpasses, i.e. they must begin and end in the same subpass.

To begin a query, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBeginQuery(
    VkCommandBuffer commandBuffer,
    VkQueryPool queryPool,
    uint32_t query,
    VkQueryControlFlags flags);
```

- commandBuffer is the command buffer into which this command will be recorded.
- queryPool is the query pool that will manage the results of the query.
- query is the query index within the query pool that will contain the results.
- flags is a bitmask of VkQueryControlFlagBits specifying constraints on the types of queries that can be performed.

If the queryType of the pool is VK_QUERY_TYPE_OCCLUSION and flags contains VK_QUERY_CONTROL_PRECISE_BIT, an implementation must return a result that matches the actual number of samples passed. This is described in more detail in Occlusion Queries.

Calling vkCmdBeginQuery is equivalent to calling vkCmdBeginQueryIndexedEXT with the index parameter set to zero.

After beginning a query, that query is considered active within the command buffer it was called in until that same query is ended. Queries active in a primary command buffer when secondary command buffers are executed are considered active for those secondary command buffers.
Valid Usage

- **VUID-vkCmdBeginQuery-None-00807**
  All queries used by the command must be unavailable

- **VUID-vkCmdBeginQuery-queryType-02804**
  The queryType used to create queryPool must not be VK_QUERY_TYPE_TIMESTAMP

- **VUID-vkCmdBeginQuery-queryType-04728**
  The queryType used to create queryPool must not be
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR or
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE.Serialization_SIZE_KHR

- **VUID-vkCmdBeginQuery-queryType-04729**
  The queryType used to create queryPool must not be
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV

- **VUID-vkCmdBeginQuery-queryType-00800**
  If the precise occlusion queries feature is not enabled, or the queryType used to create queryPool was not VK_QUERY_TYPE_OCCLUSION, flags must not contain VK_QUERY_CONTROL_PRECISE_BIT

- **VUID-vkCmdBeginQuery-query-00802**
  query must be less than the number of queries in queryPool

- **VUID-vkCmdBeginQuery-queryType-00803**
  If the queryType used to create queryPool was VK_QUERY_TYPE_OCCLUSION, the VkCommandPool that commandBuffer was allocated from must support graphics operations

- **VUID-vkCmdBeginQuery-queryType-00804**
  If the queryType used to create queryPool was VK_QUERY_TYPE_PIPELINE_STATISTICS and any of the pipelineStatistics indicate graphics operations, the VkCommandPool that commandBuffer was allocated from must support graphics operations

- **VUID-vkCmdBeginQuery-queryType-00805**
  If the queryType used to create queryPool was VK_QUERY_TYPE_PIPELINE_STATISTICS and any of the pipelineStatistics indicate compute operations, the VkCommandPool that commandBuffer was allocated from must support compute operations

- **VUID-vkCmdBeginQuery-commandBuffer-01885**
  commandBuffer must not be a protected command buffer

- **VUID-vkCmdBeginQuery-queryType-04862**
  If the queryType used to create queryPool was VK_QUERY_TYPE_VIDEO_ENCODE_BITSTREAM_BUFFER_RANGE_KHR the VkCommandPool that commandBuffer was allocated from must support video encode operations

- **VUID-vkCmdBeginQuery-queryPool-01922**
  queryPool must have been created with a queryType that differs from that of any queries that are active within commandBuffer
If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackQueries` must be supported.

If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the profiling lock must have been held before `vkBeginCommandBuffer` was called on `commandBuffer`.

If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one of the counters used to create `queryPool` was `VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR`, the query begin must be the first recorded command in `commandBuffer`.

If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and another query pool with a `queryType` `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` has been used within `commandBuffer`, its parent primary command buffer or secondary command buffer recorded within the same parent primary command buffer as `commandBuffer`, the `performanceCounterMultipleQueryPools` feature must be enabled.

If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, this command must not be recorded in a command buffer that, either directly or through secondary command buffers, also contains a `vkCmdResetQueryPool` command affecting the same query.
Valid Usage (Implicit)

- **VUID-vkCmdBeginQuery-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdBeginQuery-queryPool-parameter**
  - `queryPool` must be a valid `VkQueryPool` handle

- **VUID-vkCmdBeginQuery-flags-parameter**
  - `flags` must be a valid combination of `VkQueryControlFlagBits` values

- **VUID-vkCmdBeginQuery-commandBuffer-recording**
  - `commandBuffer` must be in the recording state

- **VUID-vkCmdBeginQuery-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- **VUID-vkCmdBeginQuery-commonparent**
  - Both of `commandBuffer`, and `queryPool` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To begin an indexed query, call:

```c
// Provided by VK_EXT_transform_feedback
void vkCmdBeginQueryIndexedEXT(  
    VkCommandBuffer                      commandBuffer,  
    VkQueryPool                           queryPool,  
    uint32_t query,                       
    VkQueryControlFlags flags,           
    uint32_t index);                      
```

- `commandBuffer` is the command buffer into which this command will be recorded.
• **queryPool** is the query pool that will manage the results of the query.

• **query** is the query index within the query pool that will contain the results.

• **flags** is a bitmask of `VkQueryControlFlagBits` specifying constraints on the types of queries that can be performed.

• **index** is the query type specific index. When the query type is `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the index represents the vertex stream.

The `vkCmdBeginQueryIndexedEXT` command operates the same as the `vkCmdBeginQuery` command, except that it also accepts a query type specific `index` parameter.
Valid Usage

- VUID-vkCmdBeginQueryIndexedEXT-None-00807
  All queries used by the command must be unavailable

- VUID-vkCmdBeginQueryIndexedEXT-queryType-02804
  The queryType used to create queryPool must not be VK_QUERY_TYPE_TIMESTAMP

- VUID-vkCmdBeginQueryIndexedEXT-queryType-04728
  The queryType used to create queryPool must not be
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR
  or
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR

- VUID-vkCmdBeginQueryIndexedEXT-queryType-04729
  The queryType used to create queryPool must not be
  VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV

- VUID-vkCmdBeginQueryIndexedEXT-queryType-00800
  If the precise occlusion queries feature is not enabled, or the queryType used to create
  queryPool was not VK_QUERY_TYPE_OCCCLUSION, flags must not contain
  VK_QUERY_CONTROL_PRECISE_BIT

- VUID-vkCmdBeginQueryIndexedEXT-query-00802
  query must be less than the number of queries in queryPool

- VUID-vkCmdBeginQueryIndexedEXT-queryType-00803
  If the queryType used to create queryPool was VK_QUERY_TYPE_OCCCLUSION, the VkCommandPool
  that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdBeginQueryIndexedEXT-queryType-00804
  If the queryType used to create queryPool was VK_QUERY_TYPE_PIPELINE_STATISTICS and any
  of the pipelineStatistics indicate graphics operations, the VkCommandPool that
  commandBuffer was allocated from must support graphics operations

- VUID-vkCmdBeginQueryIndexedEXT-queryType-00805
  If the queryType used to create queryPool was VK_QUERY_TYPE_PIPELINE_STATISTICS and any
  of the pipelineStatistics indicate compute operations, the VkCommandPool that
  commandBuffer was allocated from must support compute operations

- VUID-vkCmdBeginQueryIndexedEXT-commandBuffer-01885
  commandBuffer must not be a protected command buffer

- VUID-vkCmdBeginQueryIndexedEXT-queryPool-04753
  If the queryPool was created with the same queryType as that of another active query
  within commandBuffer, then index must not match the index used for the active query
• VUID-vkCmdBeginQueryIndexedEXT-queryType-02338
If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

• VUID-vkCmdBeginQueryIndexedEXT-queryType-02339
If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `index` parameter must be less than `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams`.

• VUID-vkCmdBeginQueryIndexedEXT-queryType-02340
If the `queryType` used to create `queryPool` was not `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` the `index` must be zero.

• VUID-vkCmdBeginQueryIndexedEXT-queryType-02341
If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` then `VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackQueries` must be supported.

• VUID-vkCmdBeginQueryIndexedEXT-queryPool-03223
If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the profiling lock must have been held before `vkBeginCommandBuffer` was called on `commandBuffer`.

• VUID-vkCmdBeginQueryIndexedEXT-queryPool-03224
If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and one of the counters used to create `queryPool` was `VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR`, the query begin must be the first recorded command in `commandBuffer`.

• VUID-vkCmdBeginQueryIndexedEXT-queryPool-03225
If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` and another query pool with a `queryType` `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR` has been used within `commandBuffer`, its parent primary command buffer or secondary command buffer recorded within the same parent primary command buffer as `commandBuffer`, the `performanceCounterMultipleQueryPools` feature must be enabled.

• VUID-vkCmdBeginQueryIndexedEXT-None-02863
If `queryPool` was created with a `queryType` of `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, this command must not be recorded in a command buffer that, either directly or through secondary command buffers, also contains a `vkCmdResetQueryPool` command affecting the same query.
Valid Usage (Implicit)

- **VUID-vkCmdBeginQueryIndexedEXT-commandBuffer-parameter**
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdBeginQueryIndexedEXT-queryPool-parameter**
  
  `queryPool` **must** be a valid `VkQueryPool` handle

- **VUID-vkCmdBeginQueryIndexedEXT-flags-parameter**
  
  `flags` **must** be a valid combination of `VkQueryControlFlagBits` values

- **VUID-vkCmdBeginQueryIndexedEXT-commandBuffer-recording**
  
  `commandBuffer` **must** be in the **recording state**

- **VUID-vkCmdBeginQueryIndexedEXT-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

- **VUID-vkCmdBeginQueryIndexedEXT-commonparent**
  
  Both of `commandBuffer`, and `queryPool` **must** have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

Bits which **can** be set in `vkCmdBeginQuery::flags`, specifying constraints on the types of queries that **can** be performed, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryControlFlagBits {
    VK_QUERY_CONTROL_PRECISE_BIT = 0x00000001,
} VkQueryControlFlagBits;
```

- **VK_QUERY_CONTROL_PRECISE_BIT** specifies the precision of occlusion queries.


// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryControlFlags;

VkQueryControlFlags is a bitmask type for setting a mask of zero or more VkQueryControlFlagBits.

To end a query after the set of desired draw or dispatch commands is executed, call:

// Provided by VK_VERSION_1_0
void vkCmdEndQuery(
    VkCommandBuffer commandBuffer,
    VkQueryPool queryPool,
    uint32_t query);

- commandBuffer is the command buffer into which this command will be recorded.
- queryPool is the query pool that is managing the results of the query.
- query is the query index within the query pool where the result is stored.

Calling vkCmdEndQuery is equivalent to calling vkCmdEndQueryIndexedEXT with the index parameter set to zero.

As queries operate asynchronously, ending a query does not immediately set the query's status to available. A query is considered finished when the final results of the query are ready to be retrieved by vkGetQueryPoolResults and vkCmdCopyQueryPoolResults, and this is when the query's status is set to available.

Once a query is ended the query must finish in finite time, unless the state of the query is changed using other commands, e.g. by issuing a reset of the query.
Valid Usage

- VUID-vkCmdEndQuery-None-01923
  All queries used by the command must be active

- VUID-vkCmdEndQuery-query-00810
  query must be less than the number of queries in queryPool

- VUID-vkCmdEndQuery-commandBuffer-01886
  commandBuffer must not be a protected command buffer

- VUID-vkCmdEndQuery-query-00812
  If vkCmdEndQuery is called within a render pass instance, the sum of query and the number of bits set in the current subpass's view mask must be less than or equal to the number of queries in queryPool

- VUID-vkCmdEndQuery-queryPool-03227
  If queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR and one or more of the counters used to create queryPool was VK_PERFORMANCE_COUNTER_SCOPE_COMMAND_BUFFER_KHR, the vkCmdEndQuery must be the last recorded command in commandBuffer

- VUID-vkCmdEndQuery-queryPool-03228
  If queryPool was created with a queryType of VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR and one or more of the counters used to create queryPool was VK_PERFORMANCE_COUNTER_SCOPE_RENDER_PASS_KHR, the vkCmdEndQuery must not be recorded within a render pass instance

Valid Usage (Implicit)

- VUID-vkCmdEndQuery-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdEndQuery-queryPool-parameter
  queryPool must be a valid VkQueryPool handle

- VUID-vkCmdEndQuery-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdEndQuery-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdEndQuery-commonparent
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To end an indexed query after the set of desired draw or dispatch commands is recorded, call:

```c
// Provided by VK_EXT_transform_feedback
void vkCmdEndQueryIndexedEXT(
    VkCommandBuffer commandBuffer,   
    VkQueryPool queryPool,          
    uint32_t query,                 
    uint32_t index);               
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `queryPool` is the query pool that is managing the results of the query.
- `query` is the query index within the query pool where the result is stored.
- `index` is the query type specific index.

The `vkCmdEndQueryIndexedEXT` command operates the same as the `vkCmdEndQuery` command, except that it also accepts a query type specific `index` parameter.
Valid Usage

• VUID-vkCmdEndQueryIndexedEXT-None-02342
  All queries used by the command must be active

• VUID-vkCmdEndQueryIndexedEXT-query-02343
  query must be less than the number of queries in queryPool

• VUID-vkCmdEndQueryIndexedEXT-commandBuffer-02344
  commandBuffer must not be a protected command buffer

• VUID-vkCmdEndQueryIndexedEXT-query-02345
  If vkCmdEndQueryIndexedEXT is called within a render pass instance, the sum of query and the number of bits set in the current subpass’s view mask must be less than or equal to the number of queries in queryPool

• VUID-vkCmdEndQueryIndexedEXT-queryType-02346
  If the queryType used to create queryPool was VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT the index parameter must be less than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams

• VUID-vkCmdEndQueryIndexedEXT-queryType-02347
  If the queryType used to create queryPool was not VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT the index must be zero

• VUID-vkCmdEndQueryIndexedEXT-commonparent
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

Valid Usage (Implicit)

• VUID-vkCmdEndQueryIndexedEXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdEndQueryIndexedEXT-queryPool-parameter
  queryPool must be a valid VkQueryPool handle

• VUID-vkCmdEndQueryIndexedEXT-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdEndQueryIndexedEXT-commandBuffer-cmddpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

An application can retrieve results either by requesting they be written into application-provided memory, or by requesting they be copied into a `VkBuffer`. In either case, the layout in memory is defined as follows:

- The first query's result is written starting at the first byte requested by the command, and each subsequent query's result begins stride bytes later.
- Occlusion queries, pipeline statistics queries, transform feedback queries, and timestamp queries store results in a tightly packed array of unsigned integers, either 32- or 64-bits as requested by the command, storing the numerical results and, if requested, the availability status.
- Performance queries store results in a tightly packed array whose type is determined by the unit member of the corresponding `VkPerformanceCounterKHR`.
- If `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT` is used, the final element of each query's result is an integer indicating whether the query's result is available, with any non-zero value indicating that it is available.
- If `VK_QUERY_RESULT_WITH_STATUS_BIT_KHR` is used, the final element of each query's result is an integer value indicating that status of the query result. Positive values indicate success, negative values indicate failure, and 0 indicates that the result is not yet available. Specific error codes are encoded in the `VkQueryResultStatusKHR` enumeration.
- Occlusion queries write one integer value - the number of samples passed. Pipeline statistics queries write one integer value for each bit that is enabled in the `pipelineStatistics` when the pool is created, and the statistics values are written in bit order starting from the least significant bit. Timestamp queries write one integer value. Performance queries write one `VkPerformanceCounterResultKHR` value for each `VkPerformanceCounterKHR` in the query. Transform feedback queries write two integers; the first integer is the number of primitives successfully written to the corresponding transform feedback buffer and the second is the number of primitives output to the vertex stream, regardless of whether they were successfully captured or not. In other words, if the transform feedback buffer was sized too small for the number of primitives output by the vertex stream, the first integer represents the number of primitives actually written and the second is the number that would have been written if all the
transform feedback buffers associated with that vertex stream were large enough.

- If more than one query is retrieved and stride is not at least as large as the size of the array of values corresponding to a single query, the values written to memory are undefined.

To retrieve status and results for a set of queries, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetQueryPoolResults(
    VkDevice                                    device,  // Logical device that owns the query pool.
    VkQueryPool                                 queryPool,  // Query pool managing the queries containing the desired results.
    uint32_t                                    firstQuery,  // Initial query index.
    uint32_t                                    queryCount,  // Number of queries to read.
    size_t                                      dataSize,  // Size in bytes of the buffer pointed to by pData.
    void*                                       pData,  // Pointer to a user-allocated buffer where the results will be written.
    VkDeviceSize                                stride,  // Stride in bytes between results for individual queries within pData.
    VkQueryResultFlags                          flags)  // Bitmask of VkQueryResultFlagBits specifying how and when results are returned.
```

- device is the logical device that owns the query pool.
- queryPool is the query pool managing the queries containing the desired results.
- firstQuery is the initial query index.
- queryCount is the number of queries to read.
- dataSize is the size in bytes of the buffer pointed to by pData.
- pData is a pointer to a user-allocated buffer where the results will be written.
- stride is the stride in bytes between results for individual queries within pData.
- flags is a bitmask of VkQueryResultFlagBits specifying how and when results are returned.

The range of queries read is defined by \([firstQuery, firstQuery + queryCount - 1]\). For pipeline statistics queries, each query index in the pool contains one integer value for each bit that is enabled in VkQueryPoolCreateInfo::pipelineStatistics when the pool is created.

If no bits are set in flags, and all requested queries are in the available state, results are written as an array of 32-bit unsigned integer values. The behavior when not all queries are available, is described below.

If VK_QUERY_RESULT_64_BIT is not set and the result overflows a 32-bit value, the value may either wrap or saturate. Similarly, if VK_QUERY_RESULT_64_BIT is set and the result overflows a 64-bit value, the value may either wrap or saturate.

If VK_QUERY_RESULT_WAIT_BIT is set, Vulkan will wait for each query to be in the available state before retrieving the numerical results for that query. In this case, vkGetQueryPoolResults is guaranteed to succeed and return VK_SUCCESS if the queries become available in a finite time (i.e. if they have been issued and not reset). If queries will never finish (e.g. due to being reset but not issued), then vkGetQueryPoolResults may not return in finite time.

If VK_QUERY_RESULT_WAIT_BIT and VK_QUERY_RESULT_PARTIAL_BIT are both not set then no result values are written to pData for queries that are in the unavailable state at the time of the call, and...
vkGetQueryPoolResults returns VK_NOT_READY. However, availability state is still written to pData for those queries if VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is set. Similarly, the status is still written to pData for those queries if VK_QUERY_RESULT_WITH_STATUS_BIT_KHR is set.

**Note**
Applications **must** take care to ensure that use of the VK_QUERY_RESULT_WAIT_BIT bit has the desired effect.

For example, if a query has been used previously and a command buffer records the commands vkCmdResetQueryPool, vkCmdBeginQuery, and vkCmdEndQuery for that query, then the query will remain in the available state until vkResetQueryPool is called or the vkCmdResetQueryPool command executes on a queue. Applications **can** use fences or events to ensure that a query has already been reset before checking for its results or availability status. Otherwise, a stale value could be returned from a previous use of the query.

The above also applies when VK_QUERY_RESULT_WAIT_BIT is used in combination with VK_QUERY_RESULT_WITH_AVAILABILITY_BIT. In this case, the returned availability status **may** reflect the result of a previous use of the query unless vkResetQueryPool is called or the vkCmdResetQueryPool command has been executed since the last use of the query.

A similar situation can arise with the VK_QUERY_RESULT_WITH_STATUS_BIT_KHR flag.

**Note**
Applications **can** double-buffer query pool usage, with a pool per frame, and reset queries at the end of the frame in which they are read.

If VK_QUERY_RESULT_PARTIAL_BIT is set, VK_QUERY_RESULT_WAIT_BIT is not set, and the query's status is unavailable, an intermediate result value between zero and the final result value is written to pData for that query.

If VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is set, the final integer value written for each query is non-zero if the query's status was available or zero if the status was unavailable. When VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is used, implementations **must** guarantee that if they return a non-zero availability value then the numerical results **must** be valid, assuming the results are not reset by a subsequent command.

**Note**
Satisfying this guarantee **may** require careful ordering by the application, e.g. to read the availability status before reading the results.

If VK_QUERY_RESULT_WITH_STATUS_BIT_KHR is set, the final integer value written for each query indicates whether the result is available or not, and whether an error occurred. A value of zero indicates that the results are not yet available. Positive values indicate that the operations within the query completed successfully, and the query results are valid. Negative values indicate that the operations within the query completed unsuccessfully.
Specific result codes are defined by the VkQueryResultStatusKHR enumeration.

**Valid Usage**

- **VUID-vkGetQueryPoolResults-firstQuery-00813**
  
  firstQuery must be less than the number of queries in queryPool

- **VUID-vkGetQueryPoolResults-flags-02828**
  
  If VK_QUERY_RESULT_64_BIT is not set in flags and the queryType used to create queryPool was not VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, then pData and stride must be multiples of 4

- **VUID-vkGetQueryPoolResults-flags-00815**
  
  If VK_QUERY_RESULT_64_BIT is set in flags then pData and stride must be multiples of 8

- **VUID-vkGetQueryPoolResults-queryType-03229**
  
  If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, then pData and stride must be multiples of the size of VkPerformanceCounterResultKHR

- **VUID-vkGetQueryPoolResults-queryType-04519**
  
  If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, then stride must be large enough to contain VkQueryPoolPerformanceCreateInfoKHR::counterIndexCount used to create queryPool times the size of VkPerformanceCounterResultKHR.

- **VUID-vkGetQueryPoolResults-firstQuery-00816**
  
  The sum of firstQuery and queryCount must be less than or equal to the number of queries in queryPool

- **VUID-vkGetQueryPoolResults-dataSize-00817**
  
  dataSize must be large enough to contain the result of each query, as described here

- **VUID-vkGetQueryPoolResults-queryType-00818**
  
  If the queryType used to create queryPool was VK_QUERY_TYPE_TIMESTAMP, flags must not contain VK_QUERY_RESULT_PARTIAL_BIT

- **VUID-vkGetQueryPoolResults-queryType-03230**
  
  If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, flags must not contain VK_QUERY_RESULT_WITH_AVAILABILITY_BIT, VK_QUERY_RESULT_PARTIAL_BIT or VK_QUERY_RESULT_64_BIT

- **VUID-vkGetQueryPoolResults-queryType-03231**
  
  If the queryType used to create queryPool was VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR, the queryPool must have been recorded once for each pass as retrieved via a call to vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR

- **VUID-vkGetQueryPoolResults-queryType-04810**
  
  If the queryType used to create queryPool was VK_QUERY_TYPE_RESULT_STATUS_ONLY_KHR, flags must include VK_QUERY_RESULT_WITH_STATUS_BIT_KHR

- **VUID-vkGetQueryPoolResults-flags-04811**
  
  If flags includes VK_QUERY_RESULT_WITH_STATUS_BIT_KHR, it must not include VK_QUERY_RESULT_WITH_AVAILABILITY_BIT
Valid Usage (Implicit)

- VUID-vkGetQueryPoolResults-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetQueryPoolResults-queryPool-parameter
queryPool must be a valid VkQueryPool handle

- VUID-vkGetQueryPoolResults-pData-parameter
pData must be a valid pointer to an array of dataSize bytes

- VUID-vkGetQueryPoolResults-flags-parameter
flags must be a valid combination of VkQueryResultFlagBits values

- VUID-vkGetQueryPoolResults-dataSize-arraylength
dataSize must be greater than 0

- VUID-vkGetQueryPoolResults-queryPool-parent
queryPool must have been created, allocated, or retrieved from device

Return Codes

Success
- VK_SUCCESS
- VK_NOT_READY

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

Bits which can be set in vkGetQueryPoolResults::flags and vkCmdCopyQueryPoolResults::flags, specifying how and when results are returned, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryResultFlagBits {
    VK_QUERY_RESULT_64_BIT = 0x00000001,
    VK_QUERY_RESULT_WAIT_BIT = 0x00000002,
    VK_QUERY_RESULT_WITH_AVAILABILITY_BIT = 0x00000004,
    VK_QUERY_RESULT_PARTIAL_BIT = 0x00000008,
    #ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_KHR_video_queue
    VK_QUERY_RESULT_WITH_STATUS_BIT_KHR = 0x00000010,
    #endif
} VkQueryResultFlagBits;
```

- VK_QUERY_RESULT_64_BIT specifies the results will be written as an array of 64-bit unsigned
integer values. If this bit is not set, the results will be written as an array of 32-bit unsigned integer values.

- **VK_QUERY_RESULT_WAIT_BIT** specifies that Vulkan will wait for each query’s status to become available before retrieving its results.
- **VK_QUERY_RESULT_WITH_AVAILABILITY_BIT** specifies that the availability status accompanies the results.
- **VK_QUERY_RESULT_PARTIAL_BIT** specifies that returning partial results is acceptable.
- **VK_QUERY_RESULT_WITH_STATUS_BIT_KHR** specifies that the last value returned in the query is a VkQueryResultStatusKHR value.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryResultFlags;
```

**VkQueryResultFlags** is a bitmask type for setting a mask of zero or more VkQueryResultFlagBits.

Specific status codes that can be returned from a query are:

```c
// Provided by VK_KHR_video_queue
typedef enum VkQueryResultStatusKHR {
  VK_QUERY_RESULT_STATUS_ERROR_KHR = -1,
  VK_QUERY_RESULT_STATUS_NOT_READY_KHR = 0,
  VK_QUERY_RESULT_STATUS_COMPLETE_KHR = 1,
} VkQueryResultStatusKHR;
```

- **VK_QUERY_RESULT_STATUS_NOT_READY_KHR** specifies that the query result is not yet available.
- **VK_QUERY_RESULT_STATUS_ERROR_KHR** specifies that operations did not complete successfully.
- **VK_QUERY_RESULT_STATUS_COMPLETE_KHR** specifies that operations completed successfully and the query result is available.

To copy query statuses and numerical results directly to buffer memory, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyQueryPoolResults(
  VkCommandBuffer                             commandBuffer,
  VkQueryPool                                 queryPool,
  uint32_t                                    firstQuery,
  uint32_t                                    queryCount,
  VkBuffer                                    dstBuffer,
  VkDeviceSize                                dstOffset,
  VkDeviceSize                                stride,
  VkQueryResultFlags                          flags);
```

- **commandBuffer** is the command buffer into which this command will be recorded.
- **queryPool** is the query pool managing the queries containing the desired results.
• **firstQuery** is the initial query index.
• **queryCount** is the number of queries. **firstQuery** and **queryCount** together define a range of queries.
• **dstBuffer** is a **VkBuffer** object that will receive the results of the copy command.
• **dstOffset** is an offset into **dstBuffer**.
• **stride** is the stride in bytes between results for individual queries within **dstBuffer**. The required size of the backing memory for **dstBuffer** is determined as described above for **vkGetQueryPoolResults**.
• **flags** is a bitmask of **VkQueryResultFlagBits** specifying how and when results are returned.

**vkCmdCopyQueryPoolResults** is guaranteed to see the effect of previous uses of **vkCmdResetQueryPool** in the same queue, without any additional synchronization. Thus, the results will always reflect the most recent use of the query.

**flags** has the same possible values described above for the **flags** parameter of **vkGetQueryPoolResults**, but the different style of execution causes some subtle behavioral differences. Because **vkCmdCopyQueryPoolResults** executes in order with respect to other query commands, there is less ambiguity about which use of a query is being requested.

Results for all requested occlusion queries, pipeline statistics queries, transform feedback queries, and timestamp queries are written as 64-bit unsigned integer values if **VK_QUERY_RESULT_64_BIT** is set or 32-bit unsigned integer values otherwise. Performance queries store results in a tightly packed array whose type is determined by the **unit** member of the corresponding **VkPerformanceCounterKHR**.

If neither of **VK_QUERY_RESULT_WAIT_BIT** and **VK_QUERY_RESULT_WITH_AVAILABILITY_BIT** are set, results are only written out for queries in the available state.

If **VK_QUERY_RESULT_WAIT_BIT** is set, the implementation will wait for each query's status to be in the available state before retrieving the numerical results for that query. This is guaranteed to reflect the most recent use of the query on the same queue, assuming that the query is not being simultaneously used by other queues. If the query does not become available in a finite amount of time (e.g. due to not issuing a query since the last reset), a **VK_ERROR_DEVICE_LOST** error may occur.

Similarly, if **VK_QUERY_RESULT_WITH_AVAILABILITY_BIT** is set and **VK_QUERY_RESULT_WAIT_BIT** is not set, the availability is guaranteed to reflect the most recent use of the query on the same queue, assuming that the query is not being simultaneously used by other queues. As with **vkGetQueryPoolResults**, implementations must guarantee that if they return a non-zero availability value, then the numerical results are valid.

If **VK_QUERY_RESULT_PARTIAL_BIT** is set, **VK_QUERY_RESULT_WAIT_BIT** is not set, and the query's status is unavailable, an intermediate result value between zero and the final result value is written for that query.

**VK_QUERY_RESULT_PARTIAL_BIT** must not be used if the pool’s **queryType** is **VK_QUERY_TYPE_TIMESTAMP**.

**vkCmdCopyQueryPoolResults** is considered to be a transfer operation, and its writes to buffer memory must be synchronized using **VK_PIPELINE_STAGE_TRANSFER_BIT** and **VK_ACCESS_TRANSFER_WRITE_BIT**.
before using the results.
Valid Usage

- **VUID-vkCmdCopyQueryPoolResults-dstOffset-00819**
  
  `dstOffset` **must** be less than the size of `dstBuffer`.

- **VUID-vkCmdCopyQueryPoolResults-firstQuery-00820**
  
  `firstQuery` **must** be less than the number of queries in `queryPool`.

- **VUID-vkCmdCopyQueryPoolResults-firstQuery-00821**
  
  The sum of `firstQuery` and `queryCount` **must** be less than or equal to the number of queries in `queryPool`.

- **VUID-vkCmdCopyQueryPoolResults-flags-00822**
  
  If `VK_QUERY_RESULT_64_BIT` is not set in `flags` then `dstOffset` and `stride` **must** be multiples of 4.

- **VUID-vkCmdCopyQueryPoolResults-flags-00823**
  
  If `VK_QUERY_RESULT_64_BIT` is set in `flags` then `dstOffset` and `stride` **must** be multiples of 8.

- **VUID-vkCmdCopyQueryPoolResults-dstBuffer-00824**
  
  `dstBuffer` **must** have enough storage, from `dstOffset`, to contain the result of each query, as described [here](#).

- **VUID-vkCmdCopyQueryPoolResults-dstBuffer-00825**
  
  `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

- **VUID-vkCmdCopyQueryPoolResults-dstBuffer-00826**
  
  If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-vkCmdCopyQueryPoolResults-queryType-00827**
  
  If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_TIMESTAMP`, `flags` **must** not contain `VK_QUERY_RESULT_PARTIAL_BIT`.

- **VUID-vkCmdCopyQueryPoolResults-queryType-03232**
  
  If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, `VkPhysicalDevicePerformanceQueryPropertiesKHR::allowCommandBufferQueryCopies` **must** be `VK_TRUE`.

- **VUID-vkCmdCopyQueryPoolResults-queryType-03233**
  
  If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, `flags` **must** not contain `VK_QUERY_RESULT_WITH_AVAILABILITY_BIT`, `VK_QUERY_RESULT_PARTIAL_BIT` or `VK_QUERY_RESULT_64_BIT`.

- **VUID-vkCmdCopyQueryPoolResults-queryType-03234**
  
  If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`, the `queryPool` **must** have been submitted once for each pass as retrieved via a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

- **VUID-vkCmdCopyQueryPoolResults-queryType-02734**
  
  `vkCmdCopyQueryPoolResults` **must** not be called if the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL`.

- **VUID-vkCmdCopyQueryPoolResults-queryType-04812**
  
  `vkCmdCopyQueryPoolResults` **must** not be called if the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR`.
was VK_QUERY_TYPE_VIDEO_ENCODE_BITSTREAM_BUFFER_RANGE_KHR or VK_QUERY_TYPE_RESULT_STATUS_ONLY_KHR.

Valid Usage (Implicit)

- VUID-vkCmdCopyQueryPoolResults-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyQueryPoolResults-queryPool-parameter
  queryPool must be a valid VkQueryPool handle

- VUID-vkCmdCopyQueryPoolResults-dstBuffer-parameter
  dstBuffer must be a valid VkBuffer handle

- VUID-vkCmdCopyQueryPoolResults-flags-parameter
  flags must be a valid combination of VkQueryResultFlagBits values

- VUID-vkCmdCopyQueryPoolResults-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdCopyQueryPoolResults-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdCopyQueryPoolResults-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdCopyQueryPoolResults-commonparent
  Each of commandBuffer, dstBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics, Compute</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rendering operations such as clears, MSAA resolves, attachment load/store operations, and blits may count towards the results of queries. This behavior is implementation-dependent and may vary depending on the path used within an implementation. For example, some implementations...
have several types of clears, some of which may include vertices and some not.

### 18.3. Occlusion Queries

Occlusion queries track the number of samples that pass the per-fragment tests for a set of drawing commands. As such, occlusion queries are only available on queue families supporting graphics operations. The application can then use these results to inform future rendering decisions. An occlusion query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively. When an occlusion query begins, the count of passing samples always starts at zero. For each drawing command, the count is incremented as described in Sample Counting. If flags does not contain `VK_QUERY_CONTROL_PRECISE_BIT` an implementation may generate any non-zero result value for the query if the count of passing samples is non-zero.

#### Note

Not setting `VK_QUERY_CONTROL_PRECISE_BIT` mode may be more efficient on some implementations, and should be used where it is sufficient to know a boolean result on whether any samples passed the per-fragment tests. In this case, some implementations may only return zero or one, indifferent to the actual number of samples passing the per-fragment tests.

When an occlusion query finishes, the result for that query is marked as available. The application can then either copy the result to a buffer (via `vkCmdCopyQueryPoolResults`) or request it be put into host memory (via `vkGetQueryPoolResults`).

#### Note

If occluding geometry is not drawn first, samples can pass the depth test, but still not be visible in a final image.

### 18.4. Pipeline Statistics Queries

Pipeline statistics queries allow the application to sample a specified set of `VkPipeline` counters. These counters are accumulated by Vulkan for a set of either draw or dispatch commands while a pipeline statistics query is active. As such, pipeline statistics queries are available on queue families supporting either graphics or compute operations. The availability of pipeline statistics queries is indicated by the `pipelineStatisticsQuery` member of the `VkPhysicalDeviceFeatures` object (see `vkGetPhysicalDeviceFeatures` and `vkCreateDevice` for detecting and requesting this query type on a `VkDevice`).

A pipeline statistics query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively. When a pipeline statistics query begins, all statistics counters are set to zero. While the query is active, the pipeline type determines which set of statistics are available, but these must be configured on the query pool when it is created. If a statistic counter is issued on a command buffer that does not support the corresponding operation, the value of that counter is undefined after the query has finished. At least one statistic counter relevant to the operations supported on the recording command buffer must be enabled.

Bits which can be set to individually enable pipeline statistics counters for query pools with
VkQueryPoolCreateInfo::pipelineStatistics, and for secondary command buffers with VkCommandBufferInheritanceInfo::pipelineStatistics, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkQueryPipelineStatisticFlagBits {
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT = 0x00000001,
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT = 0x00000002,
    VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT = 0x00000004,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT = 0x00000008,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT = 0x00000010,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT = 0x00000020,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT = 0x00000040,
    VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT = 0x00000080,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT = 0x00000100,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT = 0x00000200,
    VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT = 0x00000400,
} VkQueryPipelineStatisticFlagBits;
```

- **VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT** specifies that queries managed by the pool will count the number of vertices processed by the input assembly stage. Vertices corresponding to incomplete primitives may contribute to the count.

- **VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives processed by the input assembly stage. If primitive restart is enabled, restarting the primitive topology has no effect on the count. Incomplete primitives may be counted.

- **VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of vertex shader invocations. This counter’s value is incremented each time a vertex shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of geometry shader invocations. This counter’s value is incremented each time a geometry shader is invoked. In the case of instanced geometry shaders, the geometry shader invocations count is incremented for each separate instanced invocation.

- **VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives generated by geometry shader invocations. The counter’s value is incremented each time the geometry shader emits a primitive. Restarting primitive topology using the SPIR-V instructions OpEndPrimitive or OpEndStreamPrimitive has no effect on the geometry shader output primitives count.

- **VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of primitives processed by the Primitive Clipping stage of the pipeline. The counter’s value is incremented each time a primitive reaches the primitive clipping stage.

- **VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT** specifies that queries managed by the pool will count the number of primitives output by the Primitive Clipping stage of the pipeline.
The counter's value is incremented each time a primitive passes the primitive clipping stage. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation-dependent but must satisfy the following conditions:

- If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
- Otherwise, the counter is incremented by zero or more.

- **VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of fragment shader invocations. The counter's value is incremented each time the fragment shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT** specifies that queries managed by the pool will count the number of patches processed by the tessellation control shader. The counter's value is incremented once for each patch for which a tessellation control shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of invocations of the tessellation evaluation shader. The counter's value is incremented each time the tessellation evaluation shader is invoked.

- **VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT** specifies that queries managed by the pool will count the number of compute shader invocations. The counter's value is incremented every time the compute shader is invoked. Implementations may skip the execution of certain compute shader invocations or execute additional compute shader invocations for implementation-dependent reasons as long as the results of rendering otherwise remain unchanged.

These values are intended to measure relative statistics on one implementation. Various device architectures will count these values differently. Any or all counters may be affected by the issues described in Query Operation.

**Note**

For example, tile-based rendering devices may need to replay the scene multiple times, affecting some of the counts.

If a pipeline has `rasterizerDiscardEnable` enabled, implementations may discard primitives after the final vertex processing stage. As a result, if `rasterizerDiscardEnable` is enabled, the clipping input and output primitives counters may not be incremented.

When a pipeline statistics query finishes, the result for that query is marked as available. The application can copy the result to a buffer (via `vkCmdCopyQueryPoolResults`), or request it be put into host memory (via `vkGetQueryPoolResults`).

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkQueryPipelineStatisticFlags;
```

**VkQueryPipelineStatisticFlags** is a bitmask type for setting a mask of zero or more **VkQueryPipelineStatisticFlagBits**.
18.5. Timestamp Queries

Timestamps provide applications with a mechanism for timing the execution of commands. A timestamp is an integer value generated by the VkPhysicalDevice. Unlike other queries, timestamps do not operate over a range, and so do not use vkCmdBeginQuery or vkCmdEndQuery. The mechanism is built around a set of commands that allow the application to tell the VkPhysicalDevice to write timestamp values to a query pool and then either read timestamp values on the host (using vkGetQueryPoolResults) or copy timestamp values to a VkBuffer (using vkCmdCopyQueryPoolResults). The application can then compute differences between timestamps to determine execution time.

The number of valid bits in a timestamp value is determined by the VkQueueFamilyProperties::timestampValidBits property of the queue on which the timestamp is written. Timestamps are supported on any queue which reports a non-zero value for timestampValidBits via vkGetPhysicalDeviceQueueFamilyProperties. If the timestampComputeAndGraphics limit is VK_TRUE, timestamps are supported by every queue family that supports either graphics or compute operations (see VkQueueFamilyProperties).

The number of nanoseconds it takes for a timestamp value to be incremented by 1 can be obtained from VkPhysicalDeviceLimits::timestampPeriod after a call to vkGetPhysicalDeviceProperties.

To request a timestamp, call:

// Provided by VK_KHR_synchronization2
void vkCmdWriteTimestamp2KHR(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags2KHR stage,
    VkQueryPool queryPool,
    uint32_t query);

- commandBuffer is the command buffer into which the command will be recorded.
- stage specifies a stage of the pipeline.
- queryPool is the query pool that will manage the timestamp.
- query is the query within the query pool that will contain the timestamp.

When vkCmdWriteTimestamp2KHR is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and writes a timestamp to a query pool.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations on the pipeline stage specified by pipelineStage.

The second synchronization scope includes only the timestamp write operation.

When the timestamp value is written, the availability status of the query is set to available.
Note
If an implementation is unable to detect completion and latch the timer at any specific stage of the pipeline, it may instead do so at any logically later stage.

Timestamps may only be meaningfully compared if they are written by commands submitted to the same queue.

Note
An example of such a comparison is determining the execution time of a sequence of commands.

If `vkCmdWriteTimestamp2KHR` is called while executing a render pass instance that has multiview enabled, the timestamp uses N consecutive query indices in the query pool (starting at `query`) where N is the number of bits set in the view mask of the subpass the command is executed in. The resulting query values are determined by an implementation-dependent choice of one of the following behaviors:

- The first query is a timestamp value and (if more than one bit is set in the view mask) zero is written to the remaining queries. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the difference between the first query written by each command.

- All N queries are timestamp values. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the sum of the difference between corresponding queries written by each command. The difference between corresponding queries may be the execution time of a single view.

In either case, the application can sum the differences between all N queries to determine the total execution time.
Valid Usage

- **VUID-vkCmdWriteTimestamp2KHR-stage-03929**
  If the geometry shaders feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03930**
  If the tessellation shaders feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03931**
  If the conditional rendering feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03932**
  If the fragment density map feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03933**
  If the transform feedback feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03934**
  If the mesh shaders feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03935**
  If the task shaders feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`

- **VUID-vkCmdWriteTimestamp2KHR-stage-03936**
  If the shading rate image feature is not enabled, stage must not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`

- **VUID-vkCmdWriteTimestamp2KHR-queryPool-03858**
  The synchronization2 feature must be enabled

- **VUID-vkCmdWriteTimestamp2KHR-stage-03859**
  stage must only include a single pipeline stage

- **VUID-vkCmdWriteTimestamp2KHR-stage-03860**
  stage must only include stages valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from

- **VUID-vkCmdWriteTimestamp2KHR-queryPool-03861**
  `queryPool` must have been created with a queryType of `VK_QUERY_TYPE_TIMESTAMP`

- **VUID-vkCmdWriteTimestamp2KHR-queryPool-03862**
  The query identified by `queryPool` and `query` must be unavailable

- **VUID-vkCmdWriteTimestamp2KHR-timestampValidBits-03863**
  The command pool’s queue family must support a non-zero `timestampValidBits`

- **VUID-vkCmdWriteTimestamp2KHR-None-03864**
  All queries used by the command must be unavailable

- **VUID-vkCmdWriteTimestamp2KHR-query-03865**
  If `vkCmdWriteTimestamp2KHR` is called within a render pass instance, the sum of `query` and
the number of bits set in the current subpass’s view mask must be less than or equal to the number of queries in queryPool

Valid Usage (Implicit)

- VUID-vkCmdWriteTimestamp2KHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdWriteTimestamp2KHR-stage-parameter
  stage must be a valid combination of VkPipelineStageFlagBits2KHR values
- VUID-vkCmdWriteTimestamp2KHR-stage-requiredbitmask
  stage must not be 0
- VUID-vkCmdWriteTimestamp2KHR-queryPool-parameter
  queryPool must be a valid VkQueryPool handle
- VUID-vkCmdWriteTimestamp2KHR-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdWriteTimestamp2KHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations
- VUID-vkCmdWriteTimestamp2KHR-commonparent
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Transfer Graphics Compute</td>
<td>Transfer</td>
</tr>
</tbody>
</table>

To request a timestamp, call:
// Provided by VK_VERSION_1_0

```c
void vkCmdWriteTimestamp(
    VkCommandBuffer commandBuffer,  
    VkPipelineStageFlagBits pipelineStage, 
    VkQueryPool queryPool,  
    uint32_t query);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **pipelineStage** is one of the `VkPipelineStageFlagBits`, specifying a stage of the pipeline.
- **queryPool** is the query pool that will manage the timestamp.
- **query** is the query within the query pool that will contain the timestamp.

The `vkCmdWriteTimestamp` function latches the value of the timer when all previous commands have completed executing as far as the specified pipeline stage, and writes the timestamp value to memory. When the timestamp value is written, the availability status of the query is set to available.

**Note**

If an implementation is unable to detect completion and latch the timer at any specific stage of the pipeline, it may instead do so at any logically later stage.

Timestamps may only be meaningfully compared if they are written by commands submitted to the same queue.

**Note**

An example of such a comparison is determining the execution time of a sequence of commands.

If `vkCmdWriteTimestamp` is called while executing a render pass instance that has multiview enabled, the timestamp uses N consecutive query indices in the query pool (starting at `query`) where N is the number of bits set in the view mask of the subpass the command is executed in. The resulting query values are determined by an implementation-dependent choice of one of the following behaviors:

- The first query is a timestamp value and (if more than one bit is set in the view mask) zero is written to the remaining queries. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the difference between the first query written by each command.
- All N queries are timestamp values. If two timestamps are written in the same subpass, the sum of the execution time of all views between those commands is the sum of the difference between corresponding queries written by each command. The difference between corresponding queries may be the execution time of a single view.

In either case, the application can sum the differences between all N queries to determine the total execution time.
Valid Usage

- VUID-vkCmdWriteTimestamp-pipelineStage-04074
  pipelineStage must be a valid stage for the queue family that was used to create the command pool that commandBuffer was allocated from

- VUID-vkCmdWriteTimestamp-pipelineStage-04075
  If the geometry shaders feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-vkCmdWriteTimestamp-pipelineStage-04076
  If the tessellation shaders feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdWriteTimestamp-pipelineStage-04077
  If the conditional rendering feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT

- VUID-vkCmdWriteTimestamp-pipelineStage-04078
  If the fragment density map feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT

- VUID-vkCmdWriteTimestamp-pipelineStage-04079
  If the transform feedback feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT

- VUID-vkCmdWriteTimestamp-pipelineStage-04080
  If the mesh shaders feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV or VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV

- VUID-vkCmdWriteTimestamp-pipelineStage-04081
  If the shading rate image feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV

- VUID-vkCmdWriteTimestamp-queryPool-01416
  queryPool must have been created with a queryType of VK_QUERY_TYPE_TIMESTAMP

- VUID-vkCmdWriteTimestamp-queryPool-00828
  The query identified by queryPool and query must be unavailable

- VUID-vkCmdWriteTimestamp-timestampValidBits-00829
  The command pool's queue family must support a non-zero timestampValidBits

- VUID-vkCmdWriteTimestamp-None-00830
  All queries used by the command must be unavailable

- VUID-vkCmdWriteTimestamp-query-00831
  If vkCmdWriteTimestamp is called within a render pass instance, the sum of query and the number of bits set in the current subpass's view mask must be less than or equal to the number of queries in queryPool
Valid Usage (Implicit)

- **VUID-vkCmdWriteTimestamp-commandBuffer-parameter**
  
  commandBuffer must be a valid VkCommandBuffer handle

- **VUID-vkCmdWriteTimestamp-pipelineStage-parameter**
  
  pipelineStage must be a valid VkPipelineStageFlagBits value

- **VUID-vkCmdWriteTimestamp-queryPool-parameter**
  
  queryPool must be a valid VkQueryPool handle

- **VUID-vkCmdWriteTimestamp-commandBuffer-recording**
  
  commandBuffer must be in the recording state

- **VUID-vkCmdWriteTimestamp-commandBuffer-cmdpool**
  
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- **VUID-vkCmdWriteTimestamp-commonparent**
  
  Both of commandBuffer, and queryPool must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary, Secondary</td>
<td>Both</td>
<td>Transfer, Graphics, Compute</td>
<td>Transfer</td>
</tr>
</tbody>
</table>

18.6. Performance Queries

Performance queries provide applications with a mechanism for getting performance counter information about the execution of command buffers, render passes, and commands.

Each queue family advertises the performance counters that can be queried on a queue of that family via a call to vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR. Implementations may limit access to performance counters based on platform requirements or only to specialized drivers for development purposes.
Note
This may include no performance counters being enumerated, or a reduced set. Please refer to platform-specific documentation for guidance on any such restrictions.

Performance queries use the existing `vkCmdBeginQuery` and `vkCmdEndQuery` to control what command buffers, render passes, or commands to get performance information for.

Implementations may require multiple passes where the command buffer, render passes, or commands being recorded are the same and are executed on the same queue to record performance counter data. This is achieved by submitting the same batch and providing a `VkPerformanceQuerySubmitInfoKHR` structure containing a counter pass index. The number of passes required for a given performance query pool can be queried via a call to `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`.

Note
Command buffers created with `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` must not be re-submitted. Changing command buffer usage bits may affect performance. To avoid this, the application should re-record any command buffers with the `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` when multiple counter passes are required.

Performance counter results from a performance query pool can be obtained with the command `vkGetQueryPoolResults`.

Performance query results are returned in an array of `VkPerformanceCounterResultKHR` unions containing the data associated with each counter in the query, stored in the same order as the counters supplied in `pCounterIndices` when creating the performance query. The `VkPerformanceCounterKHR::unit` enumeration specifies how to parse the counter data.

```
// Provided by VK_KHR_performance_query
typedef union VkPerformanceCounterResultKHR {
  int32_t     int32;
  int64_t     int64;
  uint32_t    uint32;
  uint64_t    uint64;
  float       float32;
  double      float64;
} VkPerformanceCounterResultKHR;
```

18.6.1. Profiling Lock
To record and submit a command buffer that contains a performance query pool the profiling lock must be held. The profiling lock must be acquired prior to any call to `vkBeginCommandBuffer` that will be using a performance query pool. The profiling lock must be held while any command buffer that contains a performance query pool is in the recording, executable, or pending state. To acquire the profiling lock, call:
device is the logical device to profile.

pInfo is a pointer to a VkAcquireProfilingLockInfoKHR structure which contains information about how the profiling is to be acquired.

Implementations may allow multiple actors to hold the profiling lock concurrently.

Valid Usage (Implicit)

- VUID-vkAcquireProfilingLockKHR-device-parameter
device must be a valid VkDevice handle
- VUID-vkAcquireProfilingLockKHR-pInfo-parameter
pInfo must be a valid pointer to a valid VkAcquireProfilingLockInfoKHR structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_TIMEOUT

The VkAcquireProfilingLockInfoKHR structure is defined as:

```
// Provided by VK_KHR_performance_query
typedef struct VkAcquireProfilingLockInfoKHR {
    VkStructureType                   sType;
    const void*                       pNext;
    VkAcquireProfilingLockFlagsKHR    flags;
    uint64_t                           timeout;
} VkAcquireProfilingLockInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is reserved for future use.
- timeout indicates how long the function waits, in nanoseconds, if the profiling lock is not available.
Valid Usage (Implicit)

- VUID-VkAcquireProfilingLockInfoKHR-sType-sType
  
sType must be VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR

- VUID-VkAcquireProfilingLockInfoKHR-pNext-pNext
  
pNext must be NULL

- VUID-VkAcquireProfilingLockInfoKHR-flags-zerobitmask
  
flags must be 0

If `timeout` is 0, `vkAcquireProfilingLockKHR` will not block while attempting to acquire the profiling lock. If `timeout` is `UINT64_MAX`, the function will not return until the profiling lock was acquired.

```c
// Provided by VK_KHR_performance_query
typedef enum VkAcquireProfilingLockFlagBitsKHR {
} VkAcquireProfilingLockFlagBitsKHR;
```

```c
// Provided by VK_KHR_performance_query
typedef VkFlags VkAcquireProfilingLockFlagsKHR;
```

`VkAcquireProfilingLockFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

To release the profiling lock, call:

```c
// Provided by VK_KHR_performance_query
void vkReleaseProfilingLockKHR(
    VkDevice device);
```

- `device` is the logical device to cease profiling on.

Valid Usage

- VUID-vkReleaseProfilingLockKHR-device-03235
  
The profiling lock of `device` must have been held via a previous successful call to `vkAcquireProfilingLockKHR`

Valid Usage (Implicit)

- VUID-vkReleaseProfilingLockKHR-device-parameter
  
`device` must be a valid `VkDevice` handle
18.7. Transform Feedback Queries

Transform feedback queries track the number of primitives attempted to be written and actually written, by the vertex stream being captured, to a transform feedback buffer. This query is updated during draw commands while transform feedback is active. The number of primitives actually written will be less than the number attempted to be written if the bound transform feedback buffer size was too small for the number of primitives actually drawn. Primitives are not written beyond the bound range of the transform feedback buffer. A transform feedback query is begun and ended by calling `vkCmdBeginQuery` and `vkCmdEndQuery`, respectively to query for vertex stream zero. `vkCmdBeginQueryIndexedEXT` and `vkCmdEndQueryIndexedEXT` can be used to begin and end transform feedback queries for any supported vertex stream. When a transform feedback query begins, the count of primitives written and primitives needed starts from zero. For each drawing command, the count is incremented as vertex attribute outputs are captured to the transform feedback buffers while transform feedback is active.

When a transform feedback query finishes, the result for that query is marked as available. The application can then either copy the result to a buffer (via `vkCmdCopyQueryPoolResults`) or request it be put into host memory (via `vkGetQueryPoolResults`).

18.8. Intel performance queries

Intel performance queries allow an application to capture performance data for a set of commands. Performance queries are used in a similar way than other types of queries. A main difference with existing queries is that the resulting data should be handed over to a library capable to produce human readable results rather than being read directly by an application.

Prior to creating a performance query pool, initialize the device for performance queries with the call:

```c
// Provided by VK_INTEL_performance_query
VkResult vkInitializePerformanceApiINTEL(
    VkDevice                                    device,
    const VkInitializePerformanceApiInfoINTEL*  pInitializeInfo);
```

- `device` is the logical device used for the queries.
- `pInitializeInfo` is a pointer to a `VkInitializePerformanceApiInfoINTEL` structure specifying initialization parameters.

**Valid Usage (Implicit)**

- VUID-vkInitializePerformanceApiINTEL-device-parameter
device must be a valid `VkDevice` handle
- VUID-vkInitializePerformanceApiINTEL-pInitializeInfo-parameter
  pInitializeInfo must be a valid pointer to a valid `VkInitializePerformanceApiInfoINTEL` structure
Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The `VkInitializePerformanceApiInfoINTEL` structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkInitializePerformanceApiInfoINTEL {
    VkStructureType sType;
    const void* pNext;
    void* pUserData;
} VkInitializePerformanceApiInfoINTEL;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `pUserData` is a pointer for application data.

**Valid Usage (Implicit)**

- VUID-VkInitializePerformanceApiInfoINTEL-sType-sType
  `sType` must be VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL
- VUID-VkInitializePerformanceApiInfoINTEL-pNext-pNext
  `pNext` must be NULL

Once performance query operations have completed, uninitialize the device for performance queries with the call:

```c
// Provided by VK_INTEL_performance_query
void vkUninitializePerformanceApiINTEL(
    VkDevice device);
```

- `device` is the logical device used for the queries.

**Valid Usage (Implicit)**

- VUID-vkUninitializePerformanceApiINTEL-device-parameter
  `device` must be a valid VkDevice handle
Some performance query features of a device can be discovered with the call:

```c
// Provided by VK_INTEL_performance_query
VkResult vkGetPerformanceParameterINTEL(
    VkDevice                                    device,
    VkPerformanceParameterTypeINTEL             parameter,
    VkPerformanceValueINTEL*                    pValue);
```

- `device` is the logical device to query.
- `parameter` is the parameter to query.
- `pValue` is a pointer to a `VkPerformanceValueINTEL` structure in which the type and value of the parameter are returned.

### Valid Usage (Implicit)

- `VUID-vkGetPerformanceParameterINTEL-device-parameter`  
  `device` must be a valid `VkDevice` handle
- `VUID-vkGetPerformanceParameterINTEL-parameter-parameter`  
  `parameter` must be a valid `VkPerformanceParameterTypeINTEL` value
- `VUID-vkGetPerformanceParameterINTEL-pValue-parameter`  
  `pValue` must be a valid pointer to a `VkPerformanceValueINTEL` structure

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_TOO_MANY_OBJECTS`
- `VK_ERROR_OUT_OF_HOST_MEMORY`

Possible values of `vkGetPerformanceParameterINTEL::parameter`, specifying a performance query feature, are:

```c
// Provided by VK_INTEL_performance_query
typedef enum VkPerformanceParameterTypeINTEL {
    VK_PERFORMANCE_PARAMETER_TYPE_HW_COUNTERS_SUPPORTED_INTEL = 0,
    VK_PERFORMANCE_PARAMETER_TYPE_STREAM_MARKER_VALID_BITS_INTEL = 1,
} VkPerformanceParameterTypeINTEL;
```

- `VK_PERFORMANCE_PARAMETER_TYPE_HW_COUNTERS_SUPPORTED_INTEL` has a boolean result which tells whether hardware counters can be captured.
- **VK_PERFORMANCE_PARAMETER_TYPE_STREAM_MARKER_VALID_BITS_INTEL** has a 32 bits integer result which tells how many bits can be written into the **VkPerformanceValueINTEL** value.

The **VkPerformanceValueINTEL** structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkPerformanceValueINTEL {
    VkPerformanceValueTypeINTEL    type;
    VkPerformanceValueDataINTEL    data;
} VkPerformanceValueINTEL;
```

- **type** is a **VkPerformanceValueTypeINTEL** value specifying the type of the returned data.
- **data** is a **VkPerformanceValueDataINTEL** union specifying the value of the returned data.

### Valid Usage (Implicit)

- **VUID-VkPerformanceValueINTEL-type-parameter**
  
  **type** must be a valid **VkPerformanceValueTypeINTEL** value

- **VUID-VkPerformanceValueINTEL-valueString-parameter**
  
  If **type** is **VK_PERFORMANCE_VALUE_TYPE_STRING_INTEL**, the **valueString** member of **data** must be a null-terminated UTF-8 string

Possible values of **VkPerformanceValueINTEL::type**, specifying the type of the data returned in **VkPerformanceValueINTEL::data**, are:

- **VK_PERFORMANCE_VALUE_TYPE_UINT32_INTEL** specifies that unsigned 32-bit integer data is returned in **data.value32**.
- **VK_PERFORMANCE_VALUE_TYPE_UINT64_INTEL** specifies that unsigned 64-bit integer data is returned in **data.value64**.
- **VK_PERFORMANCE_VALUE_TYPE_FLOAT_INTEL** specifies that floating-point data is returned in **data.valueFloat**.
- **VK_PERFORMANCE_VALUE_TYPE_BOOL_INTEL** specifies that **Bool32** data is returned in **data.valueBool**.
- **VK_PERFORMANCE_VALUE_TYPE_STRING_INTEL** specifies that a pointer to a null-terminated UTF-8 string is returned in **data.valueString**. The pointer is valid for the lifetime of the **device** parameter passed to **vkGetPerformanceParameterINTEL**.

```c
// Provided by VK_INTEL_performance_query
typedef enum VkPerformanceValueTypeINTEL {
    VK_PERFORMANCE_VALUE_TYPE_UINT32_INTEL = 0,
    VK_PERFORMANCE_VALUE_TYPE_UINT64_INTEL = 1,
    VK_PERFORMANCE_VALUE_TYPE_FLOAT_INTEL = 2,
    VK_PERFORMANCE_VALUE_TYPE_BOOL_INTEL = 3,
    VK_PERFORMANCE_VALUE_TYPE_STRING_INTEL = 4,
} VkPerformanceValueTypeINTEL;
```
The `VkPerformanceValueDataINTEL` union is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef union VkPerformanceValueDataINTEL {
    uint32_t       value32;
    uint64_t       value64;
    float          valueFloat;
    VkBool32       valueBool;
    const char*    valueString;
} VkPerformanceValueDataINTEL;
```

- `data.value32` represents 32-bit integer data.
- `data.value64` represents 64-bit integer data.
- `data.valueFloat` represents floating-point data.
- `data.valueBool` represents `Bool32` data.
- `data.valueString` represents a pointer to a null-terminated UTF-8 string.

The correct member of the union is determined by the associated `VkPerformanceValueTypeINTEL` value.

The `VkQueryPoolPerformanceQueryCreateInfoINTEL` structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkQueryPoolPerformanceQueryCreateInfoINTEL {
    VkStructureType                 sType;
    const void*                     pNext;
    VkQueryPoolSamplingModeINTEL    performanceCountersSampling;
} VkQueryPoolPerformanceQueryCreateInfoINTEL;
```

To create a pool for Intel performance queries, set `VkQueryPoolCreateInfo::queryType` to `VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL` and add a `VkQueryPoolPerformanceQueryCreateInfoINTEL` structure to the `pNext` chain of the `VkQueryPoolCreateInfo` structure.

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `performanceCountersSampling` describe how performance queries should be captured.

### Valid Usage (Implicit)

- `VUID-VkQueryPoolPerformanceQueryCreateInfoINTEL-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_QUERY_CREATE_INFO_INTEL`  
- `VUID-VkQueryPoolPerformanceQueryCreateInfoINTEL-performanceCountersSampling-parameter`  
  `performanceCountersSampling` must be a valid `VkQueryPoolSamplingModeINTEL` value.
Possible values of `VkQueryPoolPerformanceQueryCreateInfoINTEL::performanceCountersSampling` are:

```c
// Provided by VK_INTEL_performance_query
typedef enum VkQueryPoolSamplingModeINTEL {
    VK_QUERY_POOL_SAMPLING_MODE_MANUAL_INTEL = 0,
} VkQueryPoolSamplingModeINTEL;
```

- `VK_QUERY_POOL_SAMPLING_MODE_MANUAL_INTEL` is the default mode in which the application calls `vkCmdBeginQuery` and `vkCmdEndQuery` to record performance data.

To help associate query results with a particular point at which an application emitted commands, markers can be set into the command buffers with the call:

```c
// Provided by VK_INTEL_performance_query
VkResult vkCmdSetPerformanceMarkerINTEL(
    VkCommandBuffer                             commandBuffer,
    const VkPerformanceMarkerInfoINTEL*         pMarkerInfo);
```

The last marker set onto a command buffer before the end of a query will be part of the query result.

### Valid Usage (Implicit)

- **VUID-vkCmdSetPerformanceMarkerINTEL-commandBuffer-parameter**
  - `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetPerformanceMarkerINTEL-pMarkerInfo-parameter**
  - `pMarkerInfo` **must** be a valid pointer to a valid `VkPerformanceMarkerInfoINTEL` structure

- **VUID-vkCmdSetPerformanceMarkerINTEL-commandBuffer-recording**
  - `commandBuffer` **must** be in the recording state

- **VUID-vkCmdSetPerformanceMarkerINTEL-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, compute, or transfer operations

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized
## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td>Compute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
</tbody>
</table>

## Return Codes

### Success
- VK_SUCCESS

### Failure
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The `VkPerformanceMarkerInfoINTEL` structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkPerformanceMarkerInfoINTEL {
    VkStructureType    sType;
    const void*        pNext;
    uint64_t           marker;
} VkPerformanceMarkerInfoINTEL;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `marker` is the marker value that will be recorded into the opaque query results.

### Valid Usage (Implicit)

- VUID-VkPerformanceMarkerInfoINTEL-sType-sType
  - `sType` must be VK_STRUCTURE_TYPE_PERFORMANCE_MARKER_INFO_INTEL
- VUID-VkPerformanceMarkerInfoINTEL-pNext-pNext
  - `pNext` must be NULL

When monitoring the behavior of an application within the dataset generated by the entire set of applications running on the system, it is useful to identify draw calls within a potentially huge amount of performance data. To do so, application can generate stream markers that will be used to trace back a particular draw call with a particular performance data item.
VkResult vkCmdSetPerformanceStreamMarkerINTEL(VkCommandBuffer commandBuffer, const VkPerformanceStreamMarkerInfoINTEL* pMarkerInfo);

Valid Usage (Implicit)

- VUID-vkCmdSetPerformanceStreamMarkerINTEL-commandBuffer-parameter
  **commandBuffer** must be a valid **VkCommandBuffer** handle

- VUID-vkCmdSetPerformanceStreamMarkerINTEL-pMarkerInfo-parameter
  **pMarkerInfo** must be a valid pointer to a valid **VkPerformanceStreamMarkerInfoINTEL** structure

- VUID-vkCmdSetPerformanceStreamMarkerINTEL-commandBuffer-recording
  **commandBuffer** must be in the **recording state**

- VUID-vkCmdSetPerformanceStreamMarkerINTEL-commandBuffer-cmdpool
  The **VkCommandPool** that **commandBuffer** was allocated from must support graphics, compute, or transfer operations

Host Synchronization

- Host access to **commandBuffer** must be externally synchronized

- Host access to the **VkCommandPool** that **commandBuffer** was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Both</td>
<td></td>
<td>Transfer</td>
<td></td>
</tr>
</tbody>
</table>
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_TOO_MANY_OBJECTS
• VK_ERROR_OUT_OF_HOST_MEMORY

The VkPerformanceStreamMarkerInfoINTEL structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkPerformanceStreamMarkerInfoINTEL {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           marker;
} VkPerformanceStreamMarkerInfoINTEL;
```

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `marker` is the marker value that will be recorded into the reports consumed by an external application.

Valid Usage

• VUID-VkPerformanceStreamMarkerInfoINTEL-marker-02735
The value written by the application into `marker` must only used the valid bits as reported by `vkGetPerformanceParameterINTEL` with the `VK_PERFORMANCE_PARAMETER_TYPE_STREAM_MARKER_VALID_BITS_INTEL`

Valid Usage (Implicit)

• VUID-VkPerformanceStreamMarkerInfoINTEL-sType-sType
`sType` must be `VK_STRUCTURE_TYPE_PERFORMANCE_STREAM_MARKER_INFO_INTEL`

• VUID-VkPerformanceStreamMarkerInfoINTEL-pNext-pNext
`pNext` must be NULL

Some applications might want measure the effect of a set of commands with a different settings. It is possible to override a particular settings using:
// Provided by VK_INTEL_performance_query
VkResult vkCmdSetPerformanceOverrideINTEL(
    VkCommandBuffer                             commandBuffer,
    const VkPerformanceOverrideInfoINTEL*       pOverrideInfo);

- `commandBuffer` is the command buffer where the override takes place.
- `pOverrideInfo` is a pointer to a `VkPerformanceOverrideInfoINTEL` structure selecting the parameter to override.

### Valid Usage

- **VUID-vkCmdSetPerformanceOverrideINTEL-pOverrideInfo-02736**
  - `pOverrideInfo` must not be used with a `VkPerformanceOverrideTypeINTEL` that is not reported available by `vkGetPerformanceParameterINTEL`

### Valid Usage (Implicit)

- **VUID-vkCmdSetPerformanceOverrideINTEL-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- **VUID-vkCmdSetPerformanceOverrideINTEL-pOverrideInfo-parameter**
  - `pOverrideInfo` must be a valid pointer to a valid `VkPerformanceOverrideInfoINTEL` structure
- **VUID-vkCmdSetPerformanceOverrideINTEL-commandBuffer-recording**
  - `commandBuffer` must be in the recording state
- **VUID-vkCmdSetPerformanceOverrideINTEL-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, or transfer operations

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transfer</td>
<td></td>
</tr>
</tbody>
</table>

## Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The `VkPerformanceOverrideInfoINTEL` structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkPerformanceOverrideInfoINTEL {
    VkStructureType                   sType;
    const void*                       pNext;
    VkPerformanceOverrideTypeINTEL    type;
    VkBool32                          enable;
    uint64_t                          parameter;
} VkPerformanceOverrideInfoINTEL;
```

- `type` is the particular `VkPerformanceOverrideTypeINTEL` to set.
- `enable` defines whether the override is enabled.
- `parameter` is a potential required parameter for the override.

### Valid Usage (Implicit)

- VUID-VkPerformanceOverrideInfoINTEL-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_PERFORMANCE_OVERRIDE_INFO_INTEL`
- VUID-VkPerformanceOverrideInfoINTEL-pNext-pNext
  - `pNext` must be `NULL`
- VUID-VkPerformanceOverrideInfoINTEL-type-parameter
  - `type` must be a valid `VkPerformanceOverrideTypeINTEL` value
Possible values of `VkPerformanceOverrideInfoINTEL::type`, specifying performance override types, are:

```c
typedef enum VkPerformanceOverrideTypeINTEL {
    VK_PERFORMANCE_OVERRIDE_TYPE_NULL_HARDWARE_INTEL = 0,
    VK_PERFORMANCE_OVERRIDE_TYPE_FLUSH_GPU_CACHES_INTEL = 1,
} VkPerformanceOverrideTypeINTEL;
```

- `VK_PERFORMANCE_OVERRIDE_TYPE_NULL_HARDWARE_INTEL` turns all rendering operations into noop.
- `VK_PERFORMANCE_OVERRIDE_TYPE_FLUSH_GPU_CACHES_INTEL` stalls the stream of commands until all previously emitted commands have completed and all caches been flushed and invalidated.

Before submitting command buffers containing performance queries commands to a device queue, the application must acquire and set a performance query configuration. The configuration can be released once all command buffers containing performance query commands are not in a pending state.

```c
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPerformanceConfigurationINTEL)
```

To acquire a device performance configuration, call:

```c
VkResult vkAcquirePerformanceConfigurationINTEL(
    VkDevice device,
    const VkPerformanceConfigurationAcquireInfoINTEL* pAcquireInfo,
    VkPerformanceConfigurationINTEL* pConfiguration);
```

- `device` is the logical device that the performance query commands will be submitted to.
- `pAcquireInfo` is a pointer to a `VkPerformanceConfigurationAcquireInfoINTEL` structure, specifying the performance configuration to acquire.
- `pConfiguration` is a pointer to a `VkPerformanceConfigurationINTEL` handle in which the resulting configuration object is returned.
Valid Usage (Implicit)

- VUID-vkAcquirePerformanceConfigurationINTEL-device-parameter
  device must be a valid VkDevice handle

- VUID-vkAcquirePerformanceConfigurationINTEL-pAcquireInfo-parameter
  pAcquireInfo must be a valid pointer to a valid VkPerformanceConfigurationAcquireInfoINTEL structure

- VUID-vkAcquirePerformanceConfigurationINTEL-pConfiguration-parameter
  pConfiguration must be a valid pointer to a VkPerformanceConfigurationINTEL handle

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_TOO_MANY_OBJECTS
- VK_ERROR_OUT_OF_HOST_MEMORY

The VkPerformanceConfigurationAcquireInfoINTEL structure is defined as:

```c
// Provided by VK_INTEL_performance_query
typedef struct VkPerformanceConfigurationAcquireInfoINTEL {
    VkStructureType sType;
    const void* pNext;
    VkPerformanceConfigurationTypeINTEL type;
} VkPerformanceConfigurationAcquireInfoINTEL;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `type` is one of the VkPerformanceConfigurationTypeINTEL type of performance configuration that will be acquired.

Valid Usage (Implicit)

- VUID-VkPerformanceConfigurationAcquireInfoINTEL-sType-sType
  sType must be VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL

- VUID-VkPerformanceConfigurationAcquireInfoINTEL-pNext-pNext
  pNext must be NULL

- VUID-VkPerformanceConfigurationAcquireInfoINTEL-type-parameter
  type must be a valid VkPerformanceConfigurationTypeINTEL value
Possible values of `VkPerformanceConfigurationAcquireInfoINTEL::type`, specifying performance configuration types, are:

```c
// Provided by VK_INTEL_performance_query
typedef enum VkPerformanceConfigurationTypeINTEL {
    VK_PERFORMANCE_CONFIGURATION_TYPE_COMMAND_QUEUE_METRICS_DISCOVERY_ACTIVATED_INTEL = 0,
} VkPerformanceConfigurationTypeINTEL;
```

To set a performance configuration, call:

```c
// Provided by VK_INTEL_performance_query
VkResult vkQueueSetPerformanceConfigurationINTEL(
    VkQueue                                     queue,
    VkPerformanceConfigurationINTEL             configuration);
```

- `queue` is the queue on which the configuration will be used.
- `configuration` is the configuration to use.

**Valid Usage (Implicit)**

- VUID-vkQueueSetPerformanceConfigurationINTEL-queue-parameter
  `queue` must be a valid `VkQueue` handle
- VUID-vkQueueSetPerformanceConfigurationINTEL-configuration-parameter
  `configuration` must be a valid `VkPerformanceConfigurationINTEL` handle
- VUID-vkQueueSetPerformanceConfigurationINTEL-commonparent
  Both of `configuration`, and `queue` must have been created, allocated, or retrieved from the same `VkDevice`

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_TOO_MANY_OBJECTS
• VK_ERROR_OUT_OF_HOST_MEMORY

To release a device performance configuration, call:

```c
// Provided by VK_INTEL_performance_query
VkResult vkReleasePerformanceConfigurationINTEL(
    VkDevice device,
    VkPerformanceConfigurationINTEL configuration);
```

- `device` is the device associated to the configuration object to release.
- `configuration` is the configuration object to release.

Valid Usage

- VUID-vkReleasePerformanceConfigurationINTEL-configuration-02737
  `configuration` must not be released before all command buffers submitted while the configuration was set are in pending state

Valid Usage (Implicit)

- VUID-vkReleasePerformanceConfigurationINTEL-device-parameter
  `device` must be a valid `VkDevice` handle

- VUID-vkReleasePerformanceConfigurationINTEL-configuration-parameter
  If `configuration` is not `VK_NULL_HANDLE`, `configuration` must be a valid `VkPerformanceConfigurationINTEL` handle

- VUID-vkReleasePerformanceConfigurationINTEL-configuration-parent
  If `configuration` is a valid handle, it must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `configuration` must be externally synchronized
18.9. Result Status Queries

Result status queries are used for a single purpose - to check whether a set of operations has completed successfully or not, using the \texttt{VK_QUERY_RESULT_WITH_STATUS_BIT_KHR} flag.

No other data is written to such a query.

18.10. Video Encode Bitstream Buffer Range

Bitstream buffer range queries describe the range of bytes written in the bitstream buffer by video encode commands.

When an encode command is recorded within a bitstream buffer range query, two values are written to the query slot. The first value is an offset into the bitstream buffer where the encoded video data was written. This offset is an additional offset from the start of the range specified by the application. The second value is a size value describing the number of bytes written to the bitstream buffer beyond the offset.

One slot is consumed for each slice in each command recorded between a begin and end query pair.
Chapter 19. Clear Commands

19.1. Clearing Images Outside A Render Pass Instance

Color and depth/stencil images can be cleared outside a render pass instance using `vkCmdClearColorImage` or `vkCmdClearDepthStencilImage`, respectively. These commands are only allowed outside of a render pass instance.

To clear one or more subranges of a color image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearColorImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearColorValue*                    pColor,
    uint32_t                                    rangeCount,
    const VkImageSubresourceRange*              pRanges);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `image` is the image to be cleared.
- `imageLayout` specifies the current layout of the image subresource ranges to be cleared, and must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- `pColor` is a pointer to a `VkClearColorValue` structure containing the values that the image subresource ranges will be cleared to (see Clear Values below).
- `rangeCount` is the number of image subresource range structures in `pRanges`.
- `pRanges` is a pointer to an array of `VkImageSubresourceRange` structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views.

Each specified range in `pRanges` is cleared to the value specified by `pColor`. 
Valid Usage

- **VUID-vkCmdClearColorImage-image-01993**
  The **format features** of **image** must contain **VK_FORMAT_FEATURE_TRANSFER_DST_BIT**

- **VUID-vkCmdClearColorImage-image-00002**
  **image** must have been created with **VK_IMAGE_USAGE_TRANSFER_DST_BIT** usage flag

- **VUID-vkCmdClearColorImage-image-01545**
  **image** must not use a format listed in **Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views**

- **VUID-vkCmdClearColorImage-image-00003**
  If **image** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-vkCmdClearColorImage-imageLayout-00004**
  **imageLayout** must specify the layout of the image subresource ranges of **image** specified in **pRanges** at the time this command is executed on a **VkDevice**

- **VUID-vkCmdClearColorImage-imageLayout-01394**
  **imageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

- **VUID-vkCmdClearColorImage-aspectMask-02498**
  The **VkImageSubresourceRange::aspectMask** members of the elements of the **pRanges** array must each only include **VK_IMAGE_ASPECT_COLOR_BIT**

- **VUID-vkCmdClearColorImage-baseMipLevel-01470**
  The **VkImageSubresourceRange::baseMipLevel** members of the elements of the **pRanges** array must each be less than the **mipLevels** specified in **VkImageCreateInfo** when **image** was created

- **VUID-vkCmdClearColorImage-pRanges-01692**
  For each **VkImageSubresourceRange** element of **pRanges**, if the **levelCount** member is not **VK_REMAINING_MIP_LEVELS**, then **baseMipLevel + levelCount** must be less than the **mipLevels** specified in **VkImageCreateInfo** when **image** was created

- **VUID-vkCmdClearColorImage-baseArrayLayer-01472**
  The **VkImageSubresourceRange::baseArrayLayer** members of the elements of the **pRanges** array must each be less than the **arrayLayers** specified in **VkImageCreateInfo** when **image** was created

- **VUID-vkCmdClearColorImage-pRanges-01693**
  For each **VkImageSubresourceRange** element of **pRanges**, if the **layerCount** member is not **VK_REMAINING_ARRAY_LAYERS**, then **baseArrayLayer + layerCount** must be less than the **arrayLayers** specified in **VkImageCreateInfo** when **image** was created

- **VUID-vkCmdClearColorImage-image-00007**
  **image** must not have a compressed or depth/stencil format

- **VUID-vkCmdClearColorImage-commandBuffer-01805**
  If **commandBuffer** is an unprotected command buffer, then **image** must not be a protected image
If `commandBuffer` is a protected command buffer, then `image` must not be an unprotected image.

### Valid Usage (Implicit)

- **VUID-vkCmdClearColorImage-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdClearColorImage-image-parameter**
  
  `image` must be a valid `VkImage` handle.

- **VUID-vkCmdClearColorImage-imageLayout-parameter**
  
  `imageLayout` must be a valid `VkImageLayout` value.

- **VUID-vkCmdClearColorImage-pRanges-parameter**
  
  `pRanges` must be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures.

- **VUID-vkCmdClearColorImage-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdClearColorImage-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations.

- **VUID-vkCmdClearColorImage-renderpass**
  
  This command must only be called outside of a render pass instance.

- **VUID-vkCmdClearColorImage-rangeCount-arraylength**
  
  `rangeCount` must be greater than 0.

- **VUID-vkCmdClearColorImage-commonparent**
  
  Both of `commandBuffer`, and `image` must have been created, allocated, or retrieved from the same `VkDevice`.

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.
To clear one or more subranges of a depth/stencil image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearDepthStencilImage(
    VkCommandBuffer commandBuffer,    // Provided by VK_VERSION_1_0
    VkImage image,                    // Provided by VK_VERSION_1_0
    VkImageLayout imageLayout,        // Provided by VK_VERSION_1_0
    const VkClearDepthStencilValue* pDepthStencil,  // Provided by VK_VERSION_1_0
    uint32_t rangeCount,              // Provided by VK_VERSION_1_0
    const VkImageSubresourceRange* pRanges); // Provided by VK_VERSION_1_0
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `image` is the image to be cleared.
- `imageLayout` specifies the current layout of the image subresource ranges to be cleared, and must be `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`.
- `pDepthStencil` is a pointer to a `VkClearDepthStencilValue` structure containing the values that the depth and stencil image subresource ranges will be cleared to (see Clear Values below).
- `rangeCount` is the number of image subresource range structures in `pRanges`.
- `pRanges` is a pointer to an array of `VkImageSubresourceRange` structures describing a range of mipmap levels, array layers, and aspects to be cleared, as described in Image Views.
Valid Usage

- VUID-vkCmdClearDepthStencilImage-image-01994

  The *format features* of *image* must contain VK_FORMAT_FEATURE_TRANSFER_DST_BIT

- VUID-vkCmdClearDepthStencilImage-pRanges-02658

  If the *aspect* member of any element of *pRanges* includes VK_IMAGE_ASPECT_STENCIL_BIT, and *image* was created with separate stencil usage, VK_IMAGE_USAGE_TRANSFER_DST_BIT must have been included in the VkImageStencilUsageCreateInfo::stencilUsage used to create *image*

- VUID-vkCmdClearDepthStencilImage-pRanges-02659

  If the *aspect* member of any element of *pRanges* includes VK_IMAGE_ASPECT_STENCIL_BIT, and *image* was not created with separate stencil usage, VK_IMAGE_USAGE_TRANSFER_DST_BIT must have been included in the VkImageCreateInfo::usage used to create *image*

- VUID-vkCmdClearDepthStencilImage-pRanges-02660

  If the *aspect* member of any element of *pRanges* includes VK_IMAGE_ASPECT_DEPTH_BIT, VK_IMAGE_USAGE_TRANSFER_DST_BIT must have been included in the VkImageCreateInfo::usage used to create *image*

- VUID-vkCmdClearDepthStencilImage-image-00010

  If *image* is non-sparse then it must be bound completely and contiguously to a singleVkDeviceMemory object

- VUID-vkCmdClearDepthStencilImage-imageLayout-00011

  *imageLayout* must specify the layout of the image subresource ranges of *image* specified in *pRanges* at the time this command is executed on a VkDevice

- VUID-vkCmdClearDepthStencilImage-imageLayout-00012

  *imageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL

- VUID-vkCmdClearDepthStencilImage-aspectMask-02824

  The VkImageSubresourceRange::aspectMask member of each element of the *pRanges* array must not include bits other than VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT

- VUID-vkCmdClearDepthStencilImage-image-02825

  If the *image*'s format does not have a stencil component, then the VkImageSubresourceRange::aspectMask member of each element of the *pRanges* array must not include the VK_IMAGE_ASPECT_STENCIL_BIT bit

- VUID-vkCmdClearDepthStencilImage-image-02826

  If the *image*’s format does not have a depth component, then the VkImageSubresourceRange::aspectMask member of each element of the *pRanges* array must not include the VK_IMAGE_ASPECT_DEPTH_BIT bit

- VUID-vkCmdClearDepthStencilImage-baseMipLevel-01474

  The VkImageSubresourceRange::baseMipLevel members of the elements of the *pRanges* array must each be less than the *mipLevels* specified in VkImageCreateInfo when *image* was created

- VUID-vkCmdClearDepthStencilImage-pRanges-01694

  For each VkImageSubresourceRange element of *pRanges*, if the levelCount member is not
**VK_REMAINING_MIP_LEVELS**, then `baseMipLevel + levelCount` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created

- **VUID-vkCmdClearDepthStencilImage-baseArrayLayer-01476**
  The `VkImageSubresourceRange::baseArrayLayer` members of the elements of the `pRanges` array must each be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created

- **VUID-vkCmdClearDepthStencilImage-pRanges-01695**
  For each `VkImageSubresourceRange` element of `pRanges`, if the `layerCount` member is not `VK_REMAINING_ARRAY_LAYERS`, then `baseArrayLayer + layerCount` must be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created

- **VUID-vkCmdClearDepthStencilImage-image-00014**
  `image` must have a depth/stencil format

- **VUID-vkCmdClearDepthStencilImage-commandBuffer-01807**
  If `commandBuffer` is an unprotected command buffer, then `image` must not be a protected image

- **VUID-vkCmdClearDepthStencilImage-commandBuffer-01808**
  If `commandBuffer` is a protected command buffer, then `image` must not be an unprotected image
Valid Usage (Implicit)

- VUID-vkCmdClearDepthStencilImage-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdClearDepthStencilImage-image-parameter
  `image` must be a valid `VkImage` handle
- VUID-vkCmdClearDepthStencilImage-imageLayout-parameter
  `imageLayout` must be a valid `VkImageLayout` value
- VUID-vkCmdClearDepthStencilImage-pDepthStencil-parameter
  `pDepthStencil` must be a valid pointer to a valid `VkClearDepthStencilValue` structure
- VUID-vkCmdClearDepthStencilImage-pRanges-parameter
  `pRanges` must be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures
- VUID-vkCmdClearDepthStencilImage-commandBuffer-recording
  `commandBuffer` must be in the `recording` state
- VUID-vkCmdClearDepthStencilImage-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- VUID-vkCmdClearDepthStencilImage-renderpass
  This command must only be called outside of a render pass instance
- VUID-vkCmdClearDepthStencilImage-rangeCount-arraylength
  `rangeCount` must be greater than `0`
- VUID-vkCmdClearDepthStencilImage-commonparent
  Both of `commandBuffer`, and `image` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clears outside render pass instances are treated as transfer operations for the purposes of memory
19.2. Clearing Images Inside A Render Pass Instance

To clear one or more regions of color and depth/stencil attachments inside a render pass instance, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdClearAttachments(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    attachmentCount,
    const VkClearAttachment*                    pAttachments,
    uint32_t                                    rectCount,
    const VkClearRect*                          pRects);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `attachmentCount` is the number of entries in the `pAttachments` array.
- `pAttachments` is a pointer to an array of `VkClearAttachment` structures defining the attachments to clear and the clear values to use. If any attachment to be cleared in the current subpass is `VK_ATTACHMENT_UNUSED`, then the clear has no effect on that attachment.
- `rectCount` is the number of entries in the `pRects` array.
- `pRects` is a pointer to an array of `VkClearRect` structures defining regions within each selected attachment to clear.

`vkCmdClearAttachments` can clear multiple regions of each attachment used in the current subpass of a render pass instance. This command must be called only inside a render pass instance, and implicitly selects the images to clear based on the current framebuffer attachments and the command parameters.

If the render pass has a fragment density map attachment, clears follow the operations of fragment density maps as if each clear region was a primitive which generates fragments. The clear color is applied to all pixels inside each fragment's area regardless if the pixels lie outside of the clear region. Clears may have a different set of supported fragment areas than draws.

Unlike other clear commands, `vkCmdClearAttachments` executes as a drawing command, rather than a transfer command, with writes performed by it executing in rasterization order. Clears to color attachments are executed as color attachment writes, by the `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` stage. Clears to depth/stencil attachments are executed as depth writes and writes by the `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` and `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT` stages.
Valid Usage

• VUID-vkCmdClearAttachments-aspectMask-02501
If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_COLOR_BIT`, then the `colorAttachment` member of that element must either refer to a color attachment which is `VK_ATTACHMENT_UNUSED`, or must be a valid color attachment.

• VUID-vkCmdClearAttachments-aspectMask-02502
If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_DEPTH_BIT`, then the current subpass' depth/stencil attachment must either be `VK_ATTACHMENT_UNUSED`, or must have a depth component.

• VUID-vkCmdClearAttachments-aspectMask-02503
If the `aspectMask` member of any element of `pAttachments` contains `VK_IMAGE_ASPECT_STENCIL_BIT`, then the current subpass' depth/stencil attachment must either be `VK_ATTACHMENT_UNUSED`, or must have a stencil component.

• VUID-vkCmdClearAttachments-rect-02682
The `rect` member of each element of `pRects` must have an `extent.width` greater than 0.

• VUID-vkCmdClearAttachments-rect-02683
The `rect` member of each element of `pRects` must have an `extent.height` greater than 0.

• VUID-vkCmdClearAttachments-pRects-00016
The rectangular region specified by each element of `pRects` must be contained within the render area of the current render pass instance.

• VUID-vkCmdClearAttachments-pRects-00017
The layers specified by each element of `pRects` must be contained within every attachment that `pAttachments` refers to.

• VUID-vkCmdClearAttachments-layerCount-01934
The `layerCount` member of each element of `pRects` must not be 0.

• VUID-vkCmdClearAttachments-commandBuffer-02504
If `commandBuffer` is an unprotected command buffer, then each attachment to be cleared must not be a protected image.

• VUID-vkCmdClearAttachments-commandBuffer-02505
If `commandBuffer` is a protected command buffer, then each attachment to be cleared must not be an unprotected image.

• VUID-vkCmdClearAttachments-baseArrayLayer-00018
If the render pass instance this is recorded in uses multiview, then `baseArrayLayer` must be zero and `layerCount` must be one.
Valid Usage (Implicit)

- VUID-vkCmdClearAttachments-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdClearAttachments-pAttachments-parameter
  `pAttachments` must be a valid pointer to an array of `attachmentCount` valid `VkClearAttachment` structures

- VUID-vkCmdClearAttachments-pRects-parameter
  `pRects` must be a valid pointer to an array of `rectCount` `VkClearRect` structures

- VUID-vkCmdClearAttachments-commandBuffer-recording
  `commandBuffer` must be in the `recording` state

- VUID-vkCmdClearAttachments-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdClearAttachments-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdClearAttachments-attachmentCount-arraylength
  `attachmentCount` must be greater than 0

- VUID-vkCmdClearAttachments-rectCount-arraylength
  `rectCount` must be greater than 0

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkClearRect` structure is defined as:
typedef struct VkClearRect {
    VkRect2D    rect;
    uint32_t    baseArrayLayer;
    uint32_t    layerCount;
} VkClearRect;

- `rect` is the two-dimensional region to be cleared.
- `baseArrayLayer` is the first layer to be cleared.
- `layerCount` is the number of layers to clear.

The layers `[baseArrayLayer, baseArrayLayer + layerCount)` counting from the base layer of the attachment image view are cleared.

The `VkClearAttachment` structure is defined as:

typedef struct VkClearAttachment {
    VkImageAspectFlags    aspectMask;
    uint32_t              colorAttachment;
    VkClearValue          clearValue;
} VkClearAttachment;

- `aspectMask` is a mask selecting the color, depth and/or stencil aspects of the attachment to be cleared.
- `colorAttachment` is only meaningful if `VK_IMAGE_ASPECT_COLOR_BIT` is set in `aspectMask`, in which case it is an index to the `pColorAttachments` array in the `VkSubpassDescription` structure of the current subpass which selects the color attachment to clear.
- `clearValue` is the color or depth/stencil value to clear the attachment to, as described in Clear Values below.

No memory barriers are needed between `vkCmdClearAttachments` and preceding or subsequent draw or attachment clear commands in the same subpass.

The `vkCmdClearAttachments` command is not affected by the bound pipeline state.

Attachments can also be cleared at the beginning of a render pass instance by setting `loadOp` (or `stencilLoadOp`) of `VkAttachmentDescription` to `VK_ATTACHMENT_LOAD_OP_CLEAR`, as described for `vkCreateRenderPass`. 
Valid Usage

- VUID-VkClearAttachment-aspectMask-00019
  If aspectMask includes VK_IMAGE_ASPECT_COLOR_BIT, it must not include
  VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT

- VUID-VkClearAttachment-aspectMask-00020
  aspectMask must not include VK_IMAGE_ASPECT_METADATA_BIT

- VUID-VkClearAttachment-aspectMask-02246
  aspectMask must not include VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT for any index i

- VUID-VkClearAttachment-clearValue-00021
  clearValue must be a valid VkClearColorValue union

Valid Usage (Implicit)

- VUID-VkClearAttachment-aspectMask-parameter
  aspectMask must be a valid combination of VkImageAspectFlagBits values

- VUID-VkClearAttachment-aspectMask-requiredbitmask
  aspectMask must not be 0

19.3. Clear Values

The VkClearColorValue structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef union VkClearColorValue {
  float  float32[4];
  int32_t int32[4];
  uint32_t uint32[4];
} VkClearColorValue;
```

- float32 are the color clear values when the format of the image or attachment is one of the formats in the Interpretation of Numeric Format table other than signed integer (SINT) or unsigned integer (UINT). Floating point values are automatically converted to the format of the image, with the clear value being treated as linear if the image is sRGB.

- int32 are the color clear values when the format of the image or attachment is signed integer (SINT). Signed integer values are converted to the format of the image by casting to the smaller type (with negative 32-bit values mapping to negative values in the smaller type). If the integer clear value is not representable in the target type (e.g. would overflow in conversion to that type), the clear value is undefined.

- uint32 are the color clear values when the format of the image or attachment is unsigned integer (UINT). Unsigned integer values are converted to the format of the image by casting to the integer type with fewer bits.
The four array elements of the clear color map to R, G, B, and A components of image formats, in order.

If the image has more than one sample, the same value is written to all samples for any pixels being cleared.

The `VkClearDepthStencilValue` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkClearDepthStencilValue {
    float     depth;
    uint32_t   stencil;
} VkClearDepthStencilValue;
```

- `depth` is the clear value for the depth aspect of the depth/stencil attachment. It is a floating-point value which is automatically converted to the attachment's format.
- `stencil` is the clear value for the stencil aspect of the depth/stencil attachment. It is a 32-bit integer value which is converted to the attachment's format by taking the appropriate number of LSBs.

### Valid Usage

- VUID-VkClearDepthStencilValue-depth-00022

  Unless the `VK_EXT_depth_range_unrestricted` extension is enabled, `depth` must be between 0.0 and 1.0, inclusive.

The `VkClearValue` union is defined as:

```c
// Provided by VK_VERSION_1_0
typedef union VkClearValue {
    VkClearColorValue           color;
    VkClearDepthStencilValue    depthStencil;
} VkClearValue;
```

- `color` specifies the color image clear values to use when clearing a color image or attachment.
- `depthStencil` specifies the depth and stencil clear values to use when clearing a depth/stencil image or attachment.

This union is used where part of the API requires either color or depth/stencil clear values, depending on the attachment, and defines the initial clear values in the `VkRenderPassBeginInfo` structure.

**19.4. Filling Buffers**

To clear buffer data, call:
void vkCmdFillBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    VkDeviceSize size,
    uint32_t data);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `dstBuffer` is the buffer to be filled.
- `dstOffset` is the byte offset into the buffer at which to start filling, and **must** be a multiple of 4.
- `size` is the number of bytes to fill, and **must** be either a multiple of 4, or `VK_WHOLE_SIZE` to fill the range from `offset` to the end of the buffer. If `VK_WHOLE_SIZE` is used and the remaining size of the buffer is not a multiple of 4, then the nearest smaller multiple is used.
- `data` is the 4-byte word written repeatedly to the buffer to fill `size` bytes of data. The data word is written to memory according to the host endianness.

`vkCmdFillBuffer` is treated as “transfer” operation for the purposes of synchronization barriers. The `VK_BUFFER_USAGE_TRANSFER_DST_BIT` **must** be specified in `usage` of `VkBufferCreateInfo` in order for the buffer to be compatible with `vkCmdFillBuffer`. 
Valid Usage

- VUID-vkCmdFillBuffer-dstOffset-00024
dstOffset must be less than the size of dstBuffer

- VUID-vkCmdFillBuffer-dstOffset-00025
dstOffset must be a multiple of 4

- VUID-vkCmdFillBuffer-size-00026
If size is not equal to VK_WHOLE_SIZE, size must be greater than 0

- VUID-vkCmdFillBuffer-size-00027
If size is not equal to VK_WHOLE_SIZE, size must be less than or equal to the size of dstBuffer minus dstOffset

- VUID-vkCmdFillBuffer-size-00028
If size is not equal to VK_WHOLE_SIZE, size must be a multiple of 4

- VUID-vkCmdFillBuffer-dstBuffer-00029
dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag

- VUID-vkCmdFillBuffer-dstBuffer-00031
If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdFillBuffer-commandBuffer-01811
If commandBuffer is an unprotected command buffer, then dstBuffer must not be a protected buffer

- VUID-vkCmdFillBuffer-commandBuffer-01812
If commandBuffer is a protected command buffer, then dstBuffer must not be an unprotected buffer
Valid Usage (Implicit)

- VUID-vkCmdFillBuffer-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdFillBuffer-dstBuffer-parameter
  dstBuffer must be a valid VkBuffer handle
- VUID-vkCmdFillBuffer-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdFillBuffer-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics or compute operations
- VUID-vkCmdFillBuffer-renderpass
  This command must only be called outside of a render pass instance
- VUID-vkCmdFillBuffer-commonparent
  Both of commandBuffer, and dstBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer Graphics Compute</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

19.5. Updating Buffers

To update buffer data inline in a command buffer, call:
```c
void vkCmdUpdateBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    VkDeviceSize dataSize,
    const void* pData);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **dstBuffer** is a handle to the buffer to be updated.
- **dstOffset** is the byte offset into the buffer to start updating, and **must** be a multiple of 4.
- **dataSize** is the number of bytes to update, and **must** be a multiple of 4.
- **pData** is a pointer to the source data for the buffer update, and **must** be at least **dataSize** bytes in size.

**dataSize** **must** be less than or equal to 65536 bytes. For larger updates, applications **can** use buffer to buffer copies.

**Note**
Buffer updates performed with **vkCmdUpdateBuffer** first copy the data into command buffer memory when the command is recorded (which requires additional storage and may incur an additional allocation), and then copy the data from the command buffer into **dstBuffer** when the command is executed on a device.

The additional cost of this functionality compared to buffer to buffer copies means it is only recommended for very small amounts of data, and is why it is limited to only 65536 bytes.

Applications **can** work around this by issuing multiple **vkCmdUpdateBuffer** commands to different ranges of the same buffer, but it is strongly recommended that they **should** not.

The source data is copied from the user pointer to the command buffer when the command is called.

**vkCmdUpdateBuffer** is only allowed outside of a render pass. This command is treated as “transfer” operation, for the purposes of synchronization barriers. The **VK_BUFFER_USAGE_TRANSFER_DST_BIT** **must** be specified in **usage** of **VkBufferCreateInfo** in order for the buffer to be compatible with **vkCmdUpdateBuffer**.
Valid Usage

- **VUID-vkCmdUpdateBuffer-dstOffset-00032**
  \[\text{dstOffset must be less than the size of dstBuffer}\]

- **VUID-vkCmdUpdateBuffer-dataSize-00033**
  \[\text{dataSize must be less than or equal to the size of dstBuffer minus dstOffset}\]

- **VUID-vkCmdUpdateBuffer-dstBuffer-00034**
  \[\text{dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag}\]

- **VUID-vkCmdUpdateBuffer-dstBuffer-00035**
  If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdUpdateBuffer-dstOffset-00036**
  \[\text{dstOffset must be a multiple of 4}\]

- **VUID-vkCmdUpdateBuffer-dataSize-00037**
  \[\text{dataSize must be less than or equal to 65536}\]

- **VUID-vkCmdUpdateBuffer-dataSize-00038**
  \[\text{dataSize must be a multiple of 4}\]

- **VUID-vkCmdUpdateBuffer-commandBuffer-01813**
  If `commandBuffer` is an unprotected command buffer, then `dstBuffer` **must** not be a protected buffer

- **VUID-vkCmdUpdateBuffer-commandBuffer-01814**
  If `commandBuffer` is a protected command buffer, then `dstBuffer` **must** not be an unprotected buffer
Valid Usage (Implicit)

- VUID-vkCmdUpdateBuffer-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdUpdateBuffer-dstBuffer-parameter
  dstBuffer must be a valid VkBuffer handle

- VUID-vkCmdUpdateBuffer-pData-parameter
  pData must be a valid pointer to an array of dataSize bytes

- VUID-vkCmdUpdateBuffer-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdUpdateBuffer-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdUpdateBuffer-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdUpdateBuffer-dataSize-arraylength
  dataSize must be greater than 0

- VUID-vkCmdUpdateBuffer-commonparent
  Both of commandBuffer, and dstBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

Note

The pData parameter was of type uint32_t* instead of void* prior to version 1.0.19 of the Specification and VK_HEADER_VERSION 19 of the Vulkan Header Files. This was a historical anomaly, as the source data may be of other types.
Chapter 20. Copy Commands

An application can copy buffer and image data using several methods depending on the type of data transfer. Data can be copied between buffer objects with 
\texttt{vkCmdCopyBuffer2KHR} and 
\texttt{vkCmdCopyBuffer} and a portion of an image can be copied to another image with 
\texttt{vkCmdCopyImage2KHR} and 
\texttt{vkCmdCopyImage}. Image data can also be copied to and from buffer memory using 
\texttt{vkCmdCopyImageToBuffer2KHR}, 
\texttt{vkCmdCopyImageToBuffer}, 
\texttt{vkCmdCopyBufferToImage2KHR}, and 
\texttt{vkCmdCopyBufferToImage}. Image data can be blitted (with or without scaling and filtering) with 
\texttt{vkCmdBlitImage2KHR} and 
\texttt{vkCmdBlitImage}. Multisampled images can be resolved to a non-multisampled image with 
\texttt{vkCmdResolveImage2KHR} and 
\texttt{vkCmdResolveImage}.

20.1. Common Operation

The following valid usage rules apply to all copy commands:

- Copy commands must be recorded outside of a render pass instance.
- The set of all bytes bound to all the source regions must not overlap the set of all bytes bound to the destination regions.
- The set of all bytes bound to each destination region must not overlap the set of all bytes bound to another destination region.
- Copy regions must be non-empty.
- Regions must not extend outside the bounds of the buffer or image level, except that regions of compressed images can extend as far as the dimension of the image level rounded up to a complete compressed texel block.
- Source image subresources must be in either the \texttt{VK\_IMAGE\_LAYOUT\_GENERAL} or \texttt{VK\_IMAGE\_LAYOUT\_TRANSFER\_SRC\_OPTIMAL} layout. Destination image subresources must be in the \texttt{VK\_IMAGE\_LAYOUT\_SHARED\_PRESENT\_KHR}, \texttt{VK\_IMAGE\_LAYOUT\_GENERAL} or \texttt{VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL} layout. As a consequence, if an image subresource is used as both source and destination of a copy, it must be in the \texttt{VK\_IMAGE\_LAYOUT\_GENERAL} layout.
- Source images must have \texttt{VK\_FORMAT\_FEATURE\_TRANSFER\_SRC\_BIT} in their format features.
- Destination images must have \texttt{VK\_FORMAT\_FEATURE\_TRANSFER\_DST\_BIT} in their format features.
- Source buffers must have been created with the \texttt{VK\_BUFFER\_USAGE\_TRANSFER\_SRC\_BIT} usage bit enabled and destination buffers must have been created with the \texttt{VK\_BUFFER\_USAGE\_TRANSFER\_DST\_BIT} usage bit enabled.
- Source images must have been created with \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_SRC\_BIT} set in \texttt{VkImageCreateInfo::usage}
- Destination images must have been created with \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT} set in \texttt{VkImageCreateInfo::usage}
- If the stencil aspect of source image is accessed, and the source image was not created with separate stencil usage, the source image must have been created with \texttt{VK\_IMAGE\_USAGE\_TRANSFER\_SRC\_BIT} set in \texttt{VkImageCreateInfo::usage}
- If the stencil aspect of destination image is accessed, and the destination image was not created
with separate stencil usage, the destination image must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_DST_BIT} set in \texttt{VkImageCreateInfo::usage}

- If the stencil aspect of source image is accessed, and the source image was created with separate stencil usage, the source image must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_SRC_BIT} set in \texttt{VkImageStencilUsageCreateInfo::stencilUsage}

- If the stencil aspect of destination image is accessed, and the destination image was created with separate stencil usage, the destination image must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_DST_BIT} set in \texttt{VkImageStencilUsageCreateInfo::stencilUsage}

- If non-stencil aspects of a source image are accessed, the source image must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_SRC_BIT} set in \texttt{VkImageCreateInfo::usage}

- If non-stencil aspects of a destination image are accessed, the destination image must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_DST_BIT} set in \texttt{VkImageCreateInfo::usage}

All copy commands are treated as “transfer” operations for the purposes of synchronization barriers.

All copy commands that have a source format with an X component in its format description read undefined values from those bits.

All copy commands that have a destination format with an X component in its format description write undefined values to those bits.

### 20.2. Copying Data Between Buffers

To copy data between buffer objects, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyBuffer(
    VkCommandBuffer commandBuffer,
    VkBuffer srcBuffer,
    VkBuffer dstBuffer,
    uint32_t regionCount,
    const VkBufferCopy* pRegions);
```

- \texttt{commandBuffer} is the command buffer into which the command will be recorded.
- \texttt{srcBuffer} is the source buffer.
- \texttt{dstBuffer} is the destination buffer.
- \texttt{regionCount} is the number of regions to copy.
- \texttt{pRegions} is a pointer to an array of \texttt{VkBufferCopy} structures specifying the regions to copy.

Each region in \texttt{pRegions} is copied from the source buffer to the same region of the destination buffer. \texttt{srcBuffer} and \texttt{dstBuffer} can be the same buffer or alias the same memory, but the resulting values are undefined if the copy regions overlap in memory.
Valid Usage

- VUID-vkCmdCopyBuffer-commandBuffer-01822
  If `commandBuffer` is an unprotected command buffer, then `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer-commandBuffer-01823
  If `commandBuffer` is an unprotected command buffer, then `dstBuffer` must not be a protected buffer

- VUID-vkCmdCopyBuffer-commandBuffer-01824
  If `commandBuffer` is a protected command buffer, then `dstBuffer` must not be an unprotected buffer

- VUID-vkCmdCopyBuffer-srcOffset-00113
  The `srcOffset` member of each element of `pRegions` must be less than the size of `srcBuffer`

- VUID-vkCmdCopyBuffer-dstOffset-00114
  The `dstOffset` member of each element of `pRegions` must be less than the size of `dstBuffer`

- VUID-vkCmdCopyBuffer-size-00115
  The `size` member of each element of `pRegions` must be less than or equal to the size of `srcBuffer` minus `srcOffset`

- VUID-vkCmdCopyBuffer-size-00116
  The `size` member of each element of `pRegions` must be less than or equal to the size of `dstBuffer` minus `dstOffset`

- VUID-vkCmdCopyBuffer-pRegions-00117
  The union of the source regions, and the union of the destination regions, specified by the elements of `pRegions`, must not overlap in memory

- VUID-vkCmdCopyBuffer-srcBuffer-00118
  `srcBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag

- VUID-vkCmdCopyBuffer-srcBuffer-00119
  If `srcBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-vkCmdCopyBuffer-dstBuffer-00120
  `dstBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag

- VUID-vkCmdCopyBuffer-dstBuffer-00121
  If `dstBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object
Valid Usage (Implicit)

- **VUID-vkCmdCopyBuffer-commandBuffer-parameter**
  - *commandBuffer* must be a valid *VkCommandBuffer* handle

- **VUID-vkCmdCopyBuffer-srcBuffer-parameter**
  - *srcBuffer* must be a valid *VkBuffer* handle

- **VUID-vkCmdCopyBuffer-dstBuffer-parameter**
  - *dstBuffer* must be a valid *VkBuffer* handle

- **VUID-vkCmdCopyBuffer-pRegions-parameter**
  - *pRegions* must be a valid pointer to an array of *regionCount* valid *VkBufferCopy* structures

- **VUID-vkCmdCopyBuffer-commandBuffer-recording**
  - *commandBuffer* must be in the recording state

- **VUID-vkCmdCopyBuffer-commandBuffer-cmdpool**
  - The *VkCommandPool* that *commandBuffer* was allocated from must support transfer, graphics, or compute operations

- **VUID-vkCmdCopyBuffer-renderpass**
  - This command must only be called outside of a render pass instance

- **VUID-vkCmdCopyBuffer-regionCount-arraylength**
  - *regionCount* must be greater than 0

- **VUID-vkCmdCopyBuffer-commonparent**
  - Each of *commandBuffer*, *dstBuffer*, and *srcBuffer* must have been created, allocated, or retrieved from the same *VkDevice"

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized
- Host access to the *VkCommandPool* that *commandBuffer* was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The *VkBufferCopy* structure is defined as:
// Provided by VK_VERSION_1_0

typedef struct VkBufferCopy {
    VkDeviceSize srcOffset;
    VkDeviceSize dstOffset;
    VkDeviceSize size;
} VkBufferCopy;

• **srcOffset** is the starting offset in bytes from the start of **srcBuffer**.
• **dstOffset** is the starting offset in bytes from the start of **dstBuffer**.
• **size** is the number of bytes to copy.

---

### Valid Usage

- VUID-VkBufferCopy-size-01988
  The **size** must be greater than 0

---

A more extensible version of the copy buffer command is defined below.

To copy data between buffer objects, call:

// Provided by VK_KHR_copy_commands2

```c
void vkCmdCopyBuffer2KHR(
    VkCommandBuffer commandBuffer,
    const VkCopyBufferInfo2KHR* pCopyBufferInfo);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **pCopyBufferInfo** is a pointer to a **VkCopyBufferInfo2KHR** structure describing the copy parameters.

This command is functionally identical to **vkCmdCopyBuffer**, but includes extensible sub-structures that include **sType** and **pNext** parameters, allowing them to be more easily extended.

---

### Valid Usage

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-01822
  If **commandBuffer** is an unprotected command buffer, then **srcBuffer** must not be a protected buffer

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-01823
  If **commandBuffer** is an unprotected command buffer, then **dstBuffer** must not be a protected buffer

- VUID-vkCmdCopyBuffer2KHR-commandBuffer-01824
  If **commandBuffer** is a protected command buffer, then **dstBuffer** must not be an unprotected buffer
Valid Usage (Implicit)

- **VUID-vkCmdCopyBuffer2KHR-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- **VUID-vkCmdCopyBuffer2KHR-pCopyBufferInfo-parameter**
  - `pCopyBufferInfo` must be a valid pointer to a valid `VkCopyBufferInfo2KHR` structure
- **VUID-vkCmdCopyBuffer2KHR-commandBuffer-recording**
  - `commandBuffer` must be in the recording state
- **VUID-vkCmdCopyBuffer2KHR-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- **VUID-vkCmdCopyBuffer2KHR-renderpass**
  - This command must only be called outside of a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics, Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyBufferInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyBufferInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkBuffer srcBuffer;
    VkBuffer dstBuffer;
    uint32_t regionCount;
    const VkBufferCopy2KHR* pRegions;
} VkCopyBufferInfo2KHR;
```

- `sType` is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **srcBuffer** is the source buffer.
- **dstBuffer** is the destination buffer.
- **regionCount** is the number of regions to copy.
- **pRegions** is a pointer to an array of **VkBufferCopy2KHR** structures specifying the regions to copy.

Members defined by this structure with the same name as parameters in **vkCmdCopyBuffer** have the identical effect to those parameters; the child structure **VkBufferCopy2KHR** is a variant of **VkBufferCopy** which includes **sType** and **pNext** parameters, allowing it to be extended.

### Valid Usage

- **VUID-VkCopyBufferInfo2KHR-srcOffset-00113**
  The **srcOffset** member of each element of **pRegions** must be less than the size of **srcBuffer**

- **VUID-VkCopyBufferInfo2KHR-dstOffset-00114**
  The **dstOffset** member of each element of **pRegions** must be less than the size of **dstBuffer**

- **VUID-VkCopyBufferInfo2KHR-size-00115**
  The **size** member of each element of **pRegions** must be less than or equal to the size of **srcBuffer** minus **srcOffset**

- **VUID-VkCopyBufferInfo2KHR-size-00116**
  The **size** member of each element of **pRegions** must be less than or equal to the size of **dstBuffer** minus **dstOffset**

- **VUID-VkCopyBufferInfo2KHR-pRegions-00117**
  The union of the source regions, and the union of the destination regions, specified by the elements of **pRegions**, must not overlap in memory

- **VUID-VkCopyBufferInfo2KHR-srcBuffer-00118**
  **srcBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** usage flag

- **VUID-VkCopyBufferInfo2KHR-srcBuffer-00119**
  If **srcBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- **VUID-VkCopyBufferInfo2KHR-dstBuffer-00120**
  **dstBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag

- **VUID-VkCopyBufferInfo2KHR-dstBuffer-00121**
  If **dstBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object
Valid Usage (Implicit)

- **VUID-VkCopyBufferInfo2KHR-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2_KHR*

- **VUID-VkCopyBufferInfo2KHR-pNext-pNext**
  
  *pNext must be NULL*

- **VUID-VkCopyBufferInfo2KHR-srcBuffer-parameter**
  
  *srcBuffer must be a valid VkBuffer handle*

- **VUID-VkCopyBufferInfo2KHR-dstBuffer-parameter**
  
  *dstBuffer must be a valid VkBuffer handle*

- **VUID-VkCopyBufferInfo2KHR-pRegions-parameter**
  
  *pRegions must be a valid pointer to an array of regionCount valid VkBufferCopy2KHR structures*

- **VUID-VkCopyBufferInfo2KHR-regionCount-arraylength**
  
  *regionCount must be greater than 0*

- **VUID-VkCopyBufferInfo2KHR-commonparent**
  
  *Both of dstBuffer, and srcBuffer must have been created, allocated, or retrieved from the same VkDevice*

The `VkBufferCopy2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkBufferCopy2KHR {
    VkStructureType    sType;
    const void*        pNext;
    VkDeviceSize       srcOffset;
    VkDeviceSize       dstOffset;
    VkDeviceSize       size;
} VkBufferCopy2KHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **srcOffset** is the starting offset in bytes from the start of srcBuffer.
- **dstOffset** is the starting offset in bytes from the start of dstBuffer.
- **size** is the number of bytes to copy.

Valid Usage

- **VUID-VkBufferCopy2KHR-size-01988**
  
  *The size must be greater than 0*
Valid Usage (Implicit)

- VUID-VkBufferCopy2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_COPY_2_KHR
- VUID-VkBufferCopy2KHR-pNext-pNext
  pNext must be NULL

20.3. Copying Data Between Images

_vkCmdCopyImage_ performs image copies in a similar manner to a host memcpy. It does not perform general-purpose conversions such as scaling, resizing, blending, color-space conversion, or format conversions. Rather, it simply copies raw image data. _vkCmdCopyImage_ can copy between images with different formats, provided the formats are compatible as defined below.

To copy data between image objects, call:

```cpp
// Provided by VK_VERSION_1_0
void vkCmdCopyImage(
    VkCommandBuffer commandBuffer,        commandBuffer,
    VkImage srcImage,                     srcImage,         
    VkImageLayout srcImageLayout,         srcImageLayout,    
    VkImage dstImage,                     dstImage,         
    VkImageLayout dstImageLayout,         dstImageLayout,    
    uint32_t regionCount,                 regionCount,       
    const VkImageCopy* pRegions);         pRegions);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **srcImage** is the source image.
- **srcImageLayout** is the current layout of the source image subresource.
- **dstImage** is the destination image.
- **dstImageLayout** is the current layout of the destination image subresource.
- **regionCount** is the number of regions to copy.
- **pRegions** is a pointer to an array of VkImageCopy structures specifying the regions to copy.

Each region in **pRegions** is copied from the source image to the same region of the destination image. **srcImage** and **dstImage** can be the same image or alias the same memory.

The formats of **srcImage** and **dstImage** must be compatible. Formats are compatible if they share the same class, as shown in the Compatible Formats table. Depth/stencil formats must match exactly.

If the format of **srcImage** or **dstImage** is a multi-planar image format, regions of each plane to be copied must be specified separately using the **srcSubresource** and **dstSubresource** members of the VkImageCopy structure. In this case, the aspectMask of the srcSubresource or dstSubresource that refers to the multi-planar image must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT,
or VK_IMAGE_ASPECT_PLANE_2_BIT. For the purposes of vkCmdCopyImage, each plane of a multi-planar image is treated as having the format listed in Compatible formats of planes of multi-planar formats for the plane identified by the aspectMask of the corresponding subresource. This applies both to VkFormat and to coordinates used in the copy, which correspond to texels in the plane rather than how these texels map to coordinates in the image as a whole.

**Note**
For example, the VK_IMAGE_ASPECT_PLANE_1_BIT plane of a VK_FORMAT_G8_B8R8_2PLANE_420_UNORM image is compatible with an image of format VK_FORMAT_R8G8_UNORM and (less usefully) with the VK_IMAGE_ASPECT_PLANE_0_BIT plane of an image of format VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16, as each texel is 2 bytes in size.

vkCmdCopyImage allows copying between size-compatible compressed and uncompressed internal formats. Formats are size-compatible if the texel block size of the uncompressed format is equal to the texel block size of the compressed format. Such a copy does not perform on-the-fly compression or decompression. When copying from an uncompressed format to a compressed format, each texel of uncompressed data of the source image is copied as a raw value to the corresponding compressed texel block of the destination image. When copying from a compressed format to an uncompressed format, each compressed texel block of the source image is copied as a raw value to the corresponding texel of uncompressed data in the destination image. Thus, for example, it is legal to copy between a 128-bit uncompressed format and a compressed format which has a 128-bit sized compressed texel block representing 4×4 texels (using 8 bits per texel), or between a 64-bit uncompressed format and a compressed format which has a 64-bit sized compressed texel block representing 4×4 texels (using 4 bits per texel).

When copying between compressed and uncompressed formats the extent members represent the texel dimensions of the source image and not the destination. When copying from a compressed image to an uncompressed image the image texel dimensions written to the uncompressed image will be source extent divided by the compressed texel block dimensions. When copying from an uncompressed image to a compressed image the image texel dimensions written to the compressed image will be the source extent multiplied by the compressed texel block dimensions. In both cases the number of bytes read and the number of bytes written will be identical.

Copying to or from block-compressed images is typically done in multiples of the compressed texel block size. For this reason the extent must be a multiple of the compressed texel block dimension. There is one exception to this rule which is required to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions: if the srcImage is compressed, then:

- If extent.width is not a multiple of the compressed texel block width, then (extent.width + srcOffset.x) must equal the image subresource width.
- If extent.height is not a multiple of the compressed texel block height, then (extent.height + srcOffset.y) must equal the image subresource height.
- If extent.depth is not a multiple of the compressed texel block depth, then (extent.depth + srcOffset.z) must equal the image subresource depth.
Similarly, if the \texttt{dstImage} is compressed, then:

- If \texttt{extent.width} is not a multiple of the compressed texel block width, then \((\texttt{extent.width} + \texttt{dstOffset.x})\) must equal the image subresource width.
- If \texttt{extent.height} is not a multiple of the compressed texel block height, then \((\texttt{extent.height} + \texttt{dstOffset.y})\) must equal the image subresource height.
- If \texttt{extent.depth} is not a multiple of the compressed texel block depth, then \((\texttt{extent.depth} + \texttt{dstOffset.z})\) must equal the image subresource depth.

This allows the last compressed texel block of the image in each non-multiple dimension to be included as a source or destination of the copy.

“\texttt{422}” image formats that are not \textit{multi-planar} are treated as having a \(2\times1\) compressed texel block for the purposes of these rules.

\texttt{vkCmdCopyImage} can be used to copy image data between multisample images, but both images must have the same number of samples.
Valid Usage

- **VUID-vkCmdCopyImage-commandBuffer-01825**
  If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image

- **VUID-vkCmdCopyImage-commandBuffer-01826**
  If `commandBuffer` is an unprotected command buffer, then `dstImage` must not be a protected image

- **VUID-vkCmdCopyImage-commandBuffer-01827**
  If `commandBuffer` is a protected command buffer, then `dstImage` must not be an unprotected image

- **VUID-vkCmdCopyImage-pRegions-00124**
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory

- **VUID-vkCmdCopyImage-srcImage-01995**
The format features of `srcImage` must contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`

- **VUID-vkCmdCopyImage-srcImage-00126**
  `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag

- **VUID-vkCmdCopyImage-srcImage-01546**
  If `srcImage` is non-sparse then the image or disjoint plane to be copied must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdCopyImage-srcImageLayout-00128**
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- **VUID-vkCmdCopyImage-srcImageLayout-01917**
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`

- **VUID-vkCmdCopyImage-dstImage-01996**
The format features of `dstImage` must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`

- **VUID-vkCmdCopyImage-dstImage-00131**
  `dstImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- **VUID-vkCmdCopyImage-dstImage-01547**
  If `dstImage` is non-sparse then the image or disjoint plane that is the destination of the copy must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdCopyImage-dstImageLayout-00133**
  `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- **VUID-vkCmdCopyImage-dstImageLayout-01395**
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
If the \texttt{VkFormat} of each of \texttt{srcImage} and \texttt{dstImage} is not a \textit{multi-planar format}, the \texttt{VkFormat} of each of \texttt{srcImage} and \texttt{dstImage} must be compatible, as defined above

- VUID-vkCmdCopyImage-None-01549
  In a copy to or from a plane of a \textit{multi-planar image}, the \texttt{VkFormat} of the image and plane must be compatible according to the description of compatible planes for the plane being copied

- VUID-vkCmdCopyImage-srcImage-00136
  The sample count of \texttt{srcImage} and \texttt{dstImage} must match

- VUID-vkCmdCopyImage-srcSubresource-01696
  The \texttt{srcSubresource.mipLevel} member of each element of \texttt{pRegions} must be less than the \texttt{mipLevels} specified in \texttt{VkImageCreateInfo} when \texttt{srcImage} was created

- VUID-vkCmdCopyImage-dstSubresource-01697
  The \texttt{dstSubresource.mipLevel} member of each element of \texttt{pRegions} must be less than the \texttt{mipLevels} specified in \texttt{VkImageCreateInfo} when \texttt{dstImage} was created

- VUID-vkCmdCopyImage-srcSubresource-01698
  The \texttt{srcSubresource.baseArrayLayer} + \texttt{srcSubresource.layerCount} of each element of \texttt{pRegions} must be less than or equal to the \texttt{arrayLayers} specified in \texttt{VkImageCreateInfo} when \texttt{srcImage} was created

- VUID-vkCmdCopyImage-dstSubresource-01699
  The \texttt{dstSubresource.baseArrayLayer} + \texttt{dstSubresource.layerCount} of each element of \texttt{pRegions} must be less than or equal to the \texttt{arrayLayers} specified in \texttt{VkImageCreateInfo} when \texttt{dstImage} was created

- VUID-vkCmdCopyImage-srcOffset-01783
  The \texttt{srcOffset} and \texttt{extent} members of each element of \texttt{pRegions} must respect the image transfer granularity requirements of \texttt{commandBuffer}'s command pool's queue family, as described in \texttt{VkQueueFamilyProperties}

- VUID-vkCmdCopyImage-dstOffset-01784
  The \texttt{dstOffset} and \texttt{extent} members of each element of \texttt{pRegions} must respect the image transfer granularity requirements of \texttt{commandBuffer}'s command pool's queue family, as described in \texttt{VkQueueFamilyProperties}

- VUID-vkCmdCopyImage-dstImage-02542
  \texttt{dstImage} and \texttt{srcImage} must not have been created with flags containing \texttt{VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT}

- VUID-vkCmdCopyImage-srcImage-01551
  If neither \texttt{srcImage} nor \texttt{dstImage} has a \textit{multi-planar image format} then for each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} and \texttt{dstSubresource.aspectMask} must match

- VUID-vkCmdCopyImage-srcImage-01552
  If \texttt{srcImage} has a \texttt{VkFormat} with two planes then for each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} must be \texttt{VK_IMAGE_ASPECT_PLANE_0_BIT} or \texttt{VK_IMAGE_ASPECT_PLANE_1_BIT}

- VUID-vkCmdCopyImage-srcImage-01553
  If \texttt{srcImage} has a \texttt{VkFormat} with three planes then for each element of \texttt{pRegions}, \texttt{srcSubresource.aspectMask} must be \texttt{VK_IMAGE_ASPECT_PLANE_0_BIT}, \texttt{VK_IMAGE_ASPECT_PLANE_1_BIT}, or \texttt{VK_IMAGE_ASPECT_PLANE_2_BIT}
If dstImage has a VkFormat with two planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

If dstImage has a VkFormat with three planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

If srcImage has a multi-planar image format and the dstImage does not have a multi-planar image format, then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_COLOR_BIT

If dstImage has a multi-planar image format and the srcImage does not have a multi-planar image format, then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_COLOR_BIT

If srcImage is of type VK_IMAGE_TYPE_3D, then for each element of pRegions, srcSubresource.baseArrayLayer must be 0 and srcSubresource.layerCount must be 1

If dstImage is of type VK_IMAGE_TYPE_3D, then for each element of pRegions, dstSubresource.baseArrayLayer must be 0 and dstSubresource.layerCount must be 1

For each element of pRegions, srcSubresource.aspectMask must specify aspects present in srcImage

For each element of pRegions, dstSubresource.aspectMask must specify aspects present in dstImage

For each element of pRegions, srcOffset.x and (extent.width + srcOffset.x) must both be greater than or equal to 0 and less than or equal to the width of the specified srcSubresource of srcImage

For each element of pRegions, srcOffset.y and (extent.height + srcOffset.y) must both be greater than or equal to 0 and less than or equal to the height of the specified srcSubresource of srcImage

If srcImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, srcOffset.y must be 0 and extent.height must be 1

For each element of pRegions, srcOffset.z and (extent.depth + srcOffset.z) must both be greater than or equal to 0 and less than or equal to the depth of the specified srcSubresource of srcImage
If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.z` must be 0 and `extent.depth` must be 1

- VUID-vkCmdCopyImage-dstImage-01786
  If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffset.z` must be 0 and `extent.depth` must be 1

- VUID-vkCmdCopyImage-srcImage-01787
  If `srcImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffset.z` must be 0

- VUID-vkCmdCopyImage-dstImage-01788
  If `dstImage` is of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffset.z` must be 0

- VUID-vkCmdCopyImage-srcImage-01790
  If `srcImage` and `dstImage` are both of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `extent.depth` must be 1

- VUID-vkCmdCopyImage-dstImage-01792
  If `dstImage` is of type `VK_IMAGE_TYPE_2D`, and `srcImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `extent.depth` must equal `srcSubresource.layerCount`

- VUID-vkCmdCopyImage-dstOffset-00150
  For each element of `pRegions`, `dstOffset.x` and `(extent.width + dstOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `dstSubresource` of `dstImage`

- VUID-vkCmdCopyImage-dstOffset-00151
  For each element of `pRegions`, `dstOffset.y` and `(extent.height + dstOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `dstSubresource` of `dstImage`

- VUID-vkCmdCopyImage-dstImage-00152
  If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffset.y` must be 0 and `extent.height` must be 1

- VUID-vkCmdCopyImage-dstOffset-00153
  For each element of `pRegions`, `dstOffset.z` and `(extent.depth + dstOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`

- VUID-vkCmdCopyImage-srcImage-01727
  If `srcImage` is a blocked image, then for each element of `pRegions`, all members of `srcOffset` must be a multiple of the corresponding dimensions of the compressed texel block

- VUID-vkCmdCopyImage-srcImage-01728
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.width` must be a multiple of the compressed texel block width or `(extent.width + srcOffset.x)` must equal the width of the specified `srcSubresource` of `srcImage`

- VUID-vkCmdCopyImage-srcImage-01729
If srcImage is a blocked image, then for each element of pRegions, extent.height must be a multiple of the compressed texel block height or (extent.height + srcOffset.y) must equal the height of the specified srcSubresource of srcImage

- VUID-vkCmdCopyImage-srcImage-01730
  If srcImage is a blocked image, then for each element of pRegions, extent.depth must be a multiple of the compressed texel block depth or (extent.depth + srcOffset.z) must equal the depth of the specified srcSubresource of srcImage

- VUID-vkCmdCopyImage-dstImage-01731
  If dstImage is a blocked image, then for each element of pRegions, all members of dstOffset must be a multiple of the corresponding dimensions of the compressed texel block

- VUID-vkCmdCopyImage-dstImage-01732
  If dstImage is a blocked image, then for each element of pRegions, extent.width must be a multiple of the compressed texel block width or (extent.width + dstOffset.x) must equal the width of the specified dstSubresource of dstImage

- VUID-vkCmdCopyImage-dstImage-01733
  If dstImage is a blocked image, then for each element of pRegions, extent.height must be a multiple of the compressed texel block height or (extent.height + dstOffset.y) must equal the height of the specified dstSubresource of dstImage

- VUID-vkCmdCopyImage-dstImage-01734
  If dstImage is a blocked image, then for each element of pRegions, extent.depth must be a multiple of the compressed texel block depth or (extent.depth + dstOffset.z) must equal the depth of the specified dstSubresource of dstImage
Valid Usage (Implicit)

- VUID-vkCmdCopyImage-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyImage-srcImage-parameter
  
  `srcImage` must be a valid `VkImage` handle

- VUID-vkCmdCopyImage-srcImageLayout-parameter
  
  `srcImageLayout` must be a valid `VkImageLayout` value

- VUID-vkCmdCopyImage-dstImage-parameter
  
  `dstImage` must be a valid `VkImage` handle

- VUID-vkCmdCopyImage-dstImageLayout-parameter
  
  `dstImageLayout` must be a valid `VkImageLayout` value

- VUID-vkCmdCopyImage-pRegions-parameter
  
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkImageCopy` structures

- VUID-vkCmdCopyImage-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyImage-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyImage-renderpass
  
  This command must only be called outside of a render pass instance

- VUID-vkCmdCopyImage-regionCount-arraylength
  
  `regionCount` must be greater than 0

- VUID-vkCmdCopyImage-commonparent
  
  Each of `commandBuffer`, `dstImage`, and `srcImage` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
The `VkImageCopy` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageCopy {
    VkImageSubresourceLayers    srcSubresource;
    VkOffset3D                  srcOffset;
    VkImageSubresourceLayers    dstSubresource;
    VkOffset3D                  dstOffset;
    VkExtent3D                  extent;
} VkImageCopy;
```

- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively.
- `srcOffset` and `dstOffset` select the initial \( x \), \( y \), and \( z \) offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the image to copy in `width`, `height` and `depth`.

For `VK_IMAGE_TYPE_3D` images, copies are performed slice by slice starting with the \( z \) member of the `srcOffset` or `dstOffset`, and copying `depth` slices. For images with multiple layers, copies are performed layer by layer starting with the `baseArrayLayer` member of the `srcSubresource` or `dstSubresource` and copying `layerCount` layers. Image data can be copied between images with different image types. If one image is `VK_IMAGE_TYPE_3D` and the other image is `VK_IMAGE_TYPE_2D` with multiple layers, then each slice is copied to or from a different layer.

Copies involving a multi-planar image format specify the region to be copied in terms of the `plane` to be copied, not the coordinates of the multi-planar image. This means that copies accessing the R/B planes of “.422” format images must fit the copied region within half the `width` of the parent image, and that copies accessing the R/B planes of “.420” format images must fit the copied region within half the `width` and `height` of the parent image.

### Valid Usage

- **VUID-VkImageCopy-extent-00140**

  The number of slices of the `extent` (for 3D) or layers of the `srcSubresource` (for non-3D) must match the number of slices of the `extent` (for 3D) or layers of the `dstSubresource` (for non-3D)
Valid Usage (Implicit)

- VUID-VkImageCopy-srcSubresource-parameter
  
  The `srcSubresource` must be a valid `VkImageSubresourceLayers` structure.

- VUID-VkImageCopy-dstSubresource-parameter
  
  The `dstSubresource` must be a valid `VkImageSubresourceLayers` structure.

The `VkImageSubresourceLayers` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageSubresourceLayers {
    VkImageAspectFlags aspectMask;
    uint32_t mipLevel;
    uint32_t baseArrayLayer;
    uint32_t layerCount;
} VkImageSubresourceLayers;
```

- `aspectMask` is a combination of `VkImageAspectFlagBits`, selecting the color, depth and/or stencil aspects to be copied.
- `mipLevel` is the mipmap level to copy
- `baseArrayLayer` and `layerCount` are the starting layer and number of layers to copy.

Valid Usage

- VUID-VkImageSubresourceLayers-aspectMask-00167
  
  If `aspectMask` contains `VK_IMAGE_ASPECT_COLOR_BIT`, it must not contain either of `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`.

- VUID-VkImageSubresourceLayers-aspectMask-00168
  
  `aspectMask` must not contain `VK_IMAGE_ASPECT_METADATA_BIT`.

- VUID-VkImageSubresourceLayers-aspectMask-02247
  
  `aspectMask` must not include `VK_IMAGE_ASPECT_MEMORY_PLANE_i_BIT_EXT` for any index `i`.

- VUID-VkImageSubresourceLayers-layerCount-01700
  
  `layerCount` must be greater than 0.

Valid Usage (Implicit)

- VUID-VkImageSubresourceLayers-aspectMask-parameter
  
  `aspectMask` must be a valid combination of `VkImageAspectFlagBits` values.

- VUID-VkImageSubresourceLayers-aspectMask-requiredbitmask
  
  `aspectMask` must not be 0.
A more extensible version of the copy image command is defined below.

To copy data between image objects, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdCopyImage2KHR(
    VkCommandBuffer commandBuffer,        commandBuffer,
    const VkCopyImageInfo2KHR* pCopyImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyImageInfo` is a pointer to a `VkCopyImageInfo2KHR` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- **VUID-vkCmdCopyImage2KHR-commandBuffer-01825**
  - If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image

- **VUID-vkCmdCopyImage2KHR-commandBuffer-01826**
  - If `commandBuffer` is an unprotected command buffer, then `dstImage` must not be a protected image

- **VUID-vkCmdCopyImage2KHR-commandBuffer-01827**
  - If `commandBuffer` is a protected command buffer, then `dstImage` must not be an unprotected image

### Valid Usage (Implicit)

- **VUID-vkCmdCopyImage2KHR-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdCopyImage2KHR-pCopyImageInfo-parameter**
  - `pCopyImageInfo` must be a valid pointer to a valid `VkCopyImageInfo2KHR` structure

- **VUID-vkCmdCopyImage2KHR-commandBuffer-recording**
  - `commandBuffer` must be in the recording state

- **VUID-vkCmdCopyImage2KHR-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- **VUID-vkCmdCopyImage2KHR-renderpass**
  - This command must only be called outside of a render pass instance
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyImageInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyImageInfo2KHR {
    VkStructureType           sType;
    const void*               pNext;
    VkImage                   srcImage;
    VkImageLayout             srcImageLayout;
    VkImage                   dstImage;
    VkImageLayout             dstImageLayout;
    uint32_t                  regionCount;
    const VkImageCopy2KHR*    pRegions;
} VkCopyImageInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcImage` is the source image.
- `srcImageLayout` is the current layout of the source image subresource.
- `dstImage` is the destination image.
- `dstImageLayout` is the current layout of the destination image subresource.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkImageCopy2KHR` structures specifying the regions to copy.
Valid Usage

- VUID-VkCopyImageInfo2KHR-pRegions-00124
  The union of all source regions, and the union of all destination regions, specified by the elements of *pRegions*, **must** not overlap in memory

- VUID-VkCopyImageInfo2KHR-srcImage-01995
  The *format features* of *srcImage* **must** contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`

- VUID-VkCopyImageInfo2KHR-srcImage-00126
  *srcImage* **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag

- VUID-VkCopyImageInfo2KHR-srcImage-01546
  If *srcImage* is non-sparse then the image or *disjoint* plane to be copied **must** be bound completely and contiguously to a single *VkDeviceMemory* object

- VUID-VkCopyImageInfo2KHR-srcImageLayout-00128
  *srcImageLayout* **must** specify the layout of the image subresources of *srcImage* specified in *pRegions* at the time this command is executed on a *VkDevice*

- VUID-VkCopyImageInfo2KHR-srcImageLayout-01917
  *srcImageLayout* **must** be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`

- VUID-VkCopyImageInfo2KHR-dstImage-01996
  The *format features* of *dstImage* **must** contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`

- VUID-VkCopyImageInfo2KHR-dstImage-00131
  *dstImage* **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- VUID-VkCopyImageInfo2KHR-dstImage-01547
  If *dstImage* is non-sparse then the image or *disjoint* plane that is the destination of the copy **must** be bound completely and contiguously to a single *VkDeviceMemory* object

- VUID-VkCopyImageInfo2KHR-dstImageLayout-00133
  *dstImageLayout* **must** specify the layout of the image subresources of *dstImage* specified in *pRegions* at the time this command is executed on a *VkDevice*

- VUID-VkCopyImageInfo2KHR-dstImageLayout-01395
  *dstImageLayout* **must** be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`

- VUID-VkCopyImageInfo2KHR-srcImage-01548
  If the *VkFormat* of each of *srcImage* and *dstImage* is not a *multi-planar format*, the *VkFormat* of each of *srcImage* and *dstImage* **must** be compatible, as defined above

- VUID-VkCopyImageInfo2KHR-None-01549
  In a copy to or from a plane of a *multi-planar image*, the *VkFormat* of the image and plane **must** be compatible according to the *description of compatible planes* for the plane being copied

- VUID-VkCopyImageInfo2KHR-srcImage-00136
  The sample count of *srcImage* and *dstImage* **must** match

- VUID-VkCopyImageInfo2KHR-srcSubresource-01696
  The *srcSubresource.mipLevel* member of each element of *pRegions* **must** be less than the
mipLevels specified in VkImageCreateInfo when srcImage was created

- VUID-VkCopyImageInfo2KHR-dstSubresource-01697
  The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

- VUID-VkCopyImageInfo2KHR-srcSubresource-01698
  The srcSubresource.baseArrayLayer + srcSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created

- VUID-VkCopyImageInfo2KHR-dstSubresource-01699
  The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

- VUID-VkCopyImageInfo2KHR-srcOffset-01783
  The srcOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer's command pool's queue family, as described in VkQueueFamilyProperties

- VUID-VkCopyImageInfo2KHR-dstOffset-01784
  The dstOffset and extent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer's command pool's queue family, as described in VkQueueFamilyProperties

- VUID-VkCopyImageInfo2KHR-dstImage-02542
  dstImage and srcImage must not have been created with flags containing VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT

- VUID-VkCopyImageInfo2KHR-srcImage-01551
  If neither srcImage nor dstImage has a multi-planar image format then for each element of pRegions, srcSubresource.aspectMask and dstSubresource.aspectMask must match

- VUID-VkCopyImageInfo2KHR-srcImage-01552
  If srcImage has a VkFormat with two planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

- VUID-VkCopyImageInfo2KHR-srcImage-01553
  If srcImage has a VkFormat with three planes then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

- VUID-VkCopyImageInfo2KHR-dstImage-01554
  If dstImage has a VkFormat with two planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT or VK_IMAGE_ASPECT_PLANE_1_BIT

- VUID-VkCopyImageInfo2KHR-dstImage-01555
  If dstImage has a VkFormat with three planes then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT

- VUID-VkCopyImageInfo2KHR-srcImage-01556
  If srcImage has a multi-planar image format and the dstImage does not have a multi-planar
image format, then for each element of pRegions, dstSubresource.aspectMask must be VK_IMAGE_ASPECT_COLOR_BIT

- VUID-VkCopyImageInfo2KHR-dstImage-01557
  If dstImage has a multi-planar image format and the srcImage does not have a multi-planar image format, then for each element of pRegions, srcSubresource.aspectMask must be VK_IMAGE_ASPECT_COLOR_BIT

- VUID-VkCopyImageInfo2KHR-srcImage-04443
  If srcImage is of type VK_IMAGE_TYPE_3D, then for each element of pRegions, srcSubresource.baseArrayLayer must be 0 and srcSubresource.layerCount must be 1

- VUID-VkCopyImageInfo2KHR-srcImage-01785
  If srcImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, srcSubresource.baseArrayLayer must be 0 and extent.height must be 1

- VUID-VkCopyImageInfo2KHR-dstImage-01788
  If dstImage is of type VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0

- VUID-VkCopyImageInfo2KHR-dstImage-01786
  If dstImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, dstOffset.z must be 0 and extent.depth must be 1

- VUID-VkCopyImageInfo2KHR-dstImage-01787
  If dstImage is of type VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffset.z must be 0
must be 0

- VUID-VkCopyImageInfo2KHR-srcImage-01790
  If `srcImage` and `dstImage` are both of type `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `extent.depth` must be 1

- VUID-VkCopyImageInfo2KHR-srcImage-01791
  If `srcImage` is of type `VK_IMAGE_TYPE_2D`, and `dstImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `extent.depth` must equal `srcSubresource.layerCount`

- VUID-VkCopyImageInfo2KHR-dstImage-01792
  If `dstImage` is of type `VK_IMAGE_TYPE_2D`, and `srcImage` is of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `extent.depth` must equal `dstSubresource.layerCount`

- VUID-VkCopyImageInfo2KHR-dstOffset-00150
  For each element of `pRegions`, `dstOffset.x` and `(extent.width + dstOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `dstSubresource` of `dstImage`

- VUID-VkCopyImageInfo2KHR-dstOffset-00151
  For each element of `pRegions`, `dstOffset.y` and `(extent.height + dstOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `dstSubresource` of `dstImage`

- VUID-VkCopyImageInfo2KHR-dstImage-00152
  If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffset.y` must be 0 and `extent.height` must be 1

- VUID-VkCopyImageInfo2KHR-dstOffset-00153
  For each element of `pRegions`, `dstOffset.z` and `(extent.depth + dstOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`

- VUID-VkCopyImageInfo2KHR-srcImage-01727
  If `srcImage` is a blocked image, then for each element of `pRegions`, all members of `srcOffset` must be a multiple of the corresponding dimensions of the compressed texel block

- VUID-VkCopyImageInfo2KHR-srcImage-01728
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.width` must be a multiple of the compressed texel block width or `(extent.width + srcOffset.x)` must equal the width of the specified `srcSubresource` of `srcImage`

- VUID-VkCopyImageInfo2KHR-srcImage-01729
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.height` must be a multiple of the compressed texel block height or `(extent.height + srcOffset.y)` must equal the height of the specified `srcSubresource` of `srcImage`

- VUID-VkCopyImageInfo2KHR-srcImage-01730
  If `srcImage` is a blocked image, then for each element of `pRegions`, `extent.depth` must be a multiple of the compressed texel block depth or `(extent.depth + srcOffset.z)` must equal the depth of the specified `srcSubresource` of `srcImage`

- VUID-VkCopyImageInfo2KHR-dstImage-01731
  If `dstImage` is a blocked image, then for each element of `pRegions`, all members of `dstOffset` must be a multiple of the corresponding dimensions of the compressed texel block
If \texttt{dstImage} is a \texttt{blocked image}, then for each element of \texttt{pRegions}, \texttt{extent.width} must be a multiple of the compressed texel block width or \((\texttt{extent.width} + \texttt{dstOffset.x})\) must equal the width of the specified \texttt{dstSubresource} of \texttt{dstImage}.

If \texttt{dstImage} is a \texttt{blocked image}, then for each element of \texttt{pRegions}, \texttt{extent.height} must be a multiple of the compressed texel block height or \((\texttt{extent.height} + \texttt{dstOffset.y})\) must equal the height of the specified \texttt{dstSubresource} of \texttt{dstImage}.

If \texttt{dstImage} is a \texttt{blocked image}, then for each element of \texttt{pRegions}, \texttt{extent.depth} must be a multiple of the compressed texel block depth or \((\texttt{extent.depth} + \texttt{dstOffset.z})\) must equal the depth of the specified \texttt{dstSubresource} of \texttt{dstImage}.

Valid Usage (Implicit)

- \texttt{sType must be VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2_KHR}
- \texttt{pNext must be NULL}
- \texttt{srcImage must be a valid VkImage handle}
- \texttt{srcImageLayout must be a valid VkImageLayout value}
- \texttt{dstImage must be a valid VkImage handle}
- \texttt{dstImageLayout must be a valid VkImageLayout value}
- \texttt{pRegions must be a valid pointer to an array of regionCount valid VkImageCopy2KHR structures}
- \texttt{regionCount must be greater than 0}
- Both of \texttt{dstImage}, and \texttt{srcImage must have been created, allocated, or retrieved from the same VkDevice}

The \texttt{VkImageCopy2KHR} structure is defined as:
// Provided by VK_KHR_copy_commands2

typedef struct VkImageCopy2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageCopy2KHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **srcSubresource** and **dstSubresource** are **VkImageSubresourceLayers** structures specifying the image subresources of the images used for the source and destination image data, respectively.
- **srcOffset** and **dstOffset** select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
- **extent** is the size in texels of the image to copy in width, height and depth.

### Valid Usage

- VUID-VkImageCopy2KHR-extent-00140
  The number of slices of the **extent** (for 3D) or layers of the **srcSubresource** (for non-3D) **must** match the number of slices of the **extent** (for 3D) or layers of the **dstSubresource** (for non-3D)

### Valid Usage (Implicit)

- VUID-VkImageCopy2KHR-sType-sType
  **sType** **must** be **VK_STRUCTURE_TYPE_IMAGE_COPY_2_KHR**

- VUID-VkImageCopy2KHR-pNext-pNext
  **pNext** **must** be **NULL**

- VUID-VkImageCopy2KHR-srcSubresource-parameter
  **srcSubresource** **must** be a valid **VkImageSubresourceLayers** structure

- VUID-VkImageCopy2KHR-dstSubresource-parameter
  **dstSubresource** **must** be a valid **VkImageSubresourceLayers** structure

## 20.4. Copying Data Between Buffers and Images

To copy data from a buffer object to an image object, call:
// Provided by VK_VERSION_1_0
void vkCmdCopyBufferToImage(
    VkCommandBuffer commandBuffer,
    VkBuffer srcBuffer,
    VkImage dstImage,
    VkImageLayout dstImageLayout,
    uint32_t regionCount,
    const VkBufferImageCopy* pRegions);

• commandBuffer is the command buffer into which the command will be recorded.
• srcBuffer is the source buffer.
• dstImage is the destination image.
• dstImageLayout is the layout of the destination image subresources for the copy.
• regionCount is the number of regions to copy.
• pRegions is a pointer to an array of VkBufferImageCopy structures specifying the regions to copy.

Each region in pRegions is copied from the specified region of the source buffer to the specified region of the destination image.

If the format of dstImage is a multi-planar image format, regions of each plane to be a target of a copy must be specified separately using the pRegions member of the VkBufferImageCopy structure. In this case, the aspectMask of imageSubresource must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT. For the purposes of vkCmdCopyBufferToImage, each plane of a multi-planar image is treated as having the format listed in Compatible formats of planes of multi-planar formats for the plane identified by the aspectMask of the corresponding subresource. This applies both to VkFormat and to coordinates used in the copy, which correspond to texels in the plane rather than how these texels map to coordinates in the image as a whole.
Valid Usage

- VUID-vkCmdCopyBufferToImage-commandBuffer-01828
  If `commandBuffer` is an unprotected command buffer, then `srcBuffer` must not be a protected buffer

- VUID-vkCmdCopyBufferToImage-commandBuffer-01829
  If `commandBuffer` is an unprotected command buffer, then `dstImage` must not be a protected image

- VUID-vkCmdCopyBufferToImage-commandBuffer-01830
  If `commandBuffer` is a protected command buffer, then `dstImage` must not be an unprotected image

- VUID-vkCmdCopyBufferToImage-pRegions-00172
  The image region specified by each element of `pRegions` must be a region that is contained within `dstImage` if the `dstImage`'s `VkFormat` is not a multi-planar format, and must be a region that is contained within the plane being copied to if the `dstImage`'s `VkFormat` is a multi-planar format

- VUID-vkCmdCopyBufferToImage-pRegions-00171
  `srcBuffer` must be large enough to contain all buffer locations that are accessed according to `Buffer and Image Addressing`, for each element of `pRegions`

- VUID-vkCmdCopyBufferToImage-pRegions-00173
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory

- VUID-vkCmdCopyBufferToImage-srcBuffer-00174
  `srcBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag

- VUID-vkCmdCopyBufferToImage-dstImage-00177
  `dstImage` must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag

- VUID-vkCmdCopyBufferToImage-dstImage-00178
  If `dstImage` is non-sparse then it must be bound completely and continguously to a single `VkDeviceMemory` object

- VUID-vkCmdCopyBufferToImage-dstImage-00179
  `dstImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

- VUID-vkCmdCopyBufferToImage-dstImageLayout-00180
  `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-vkCmdCopyBufferToImage-dstImageLayout-01396
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
The `imageSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

The `imageSubresource.baseArrayLayer + imageSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

The `imageOffset` and `imageExtent` members of each element of `pRegions` must respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`.

`dstImage` must not have been created with flags containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.

If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT`, for each element of `pRegions`, the `aspectMask` member of `imageSubresource` must not be `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT`.

For each element of `pRegions`, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of `dstImage` where this refers to the width of the plane of the image involved in the copy in the case of a multi-planar format.

For each element of `pRegions`, `imageOffset.y` and `(imageExtent.height + imageOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `imageSubresource` of `dstImage` where this refers to the height of the plane of the image involved in the copy in the case of a multi-planar format.

If `dstImage` does not have either a depth/stencil or a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the format's texel block size.

If `dstImage` has a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in Compatible formats of planes of multi-planar formats.

If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `imageOffset.y` must be 0 and `imageExtent.height` must be 1.

For each element of `pRegions`, `imageOffset.z` and `(imageExtent.depth + imageOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `imageSubresource` of `dstImage`.
If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE}_1\text{D} \) or \( \text{VK_IMAGE_TYPE}_2\text{D} \), then for each element of \( \text{pRegions} \), imageOffset\( .z \) \textbf{must} be 0 and imageExtent\( .\text{depth} \) \textbf{must} be 1

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), bufferRowLength \textbf{must} be a multiple of the compressed texel block width

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), bufferImageHeight \textbf{must} be a multiple of the compressed texel block height

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), all members of imageOffset \textbf{must} be a multiple of the corresponding dimensions of the compressed texel block

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), bufferOffset \textbf{must} be a multiple of the compressed texel block size in bytes

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), imageExtent\( .\text{width} \) \textbf{must} be a multiple of the compressed texel block width or \((\text{imageExtent}\.\text{width} + \text{imageOffset}\.x)\) \textbf{must} equal the width of the specified imageSubresource of \( \text{dstImage} \)

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), imageExtent\( .\text{height} \) \textbf{must} be a multiple of the compressed texel block height or \((\text{imageExtent}\.\text{height} + \text{imageOffset}\.y)\) \textbf{must} equal the height of the specified imageSubresource of \( \text{dstImage} \)

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), imageExtent\( .\text{depth} \) \textbf{must} be a multiple of the compressed texel block depth or \((\text{imageExtent}\.\text{depth} + \text{imageOffset}\.z)\) \textbf{must} equal the depth of the specified imageSubresource of \( \text{dstImage} \)

For each element of \( \text{pRegions} \), imageSubresource\.aspectMask \textbf{must} specify aspects present in \( \text{dstImage} \)

If \( \text{dstImage} \) has a \textit{multi-planar format}, then for each element of \( \text{pRegions} \), imageSubresource\.aspectMask \textbf{must} be VK\_IMAGE\_ASPECT\_PLANE\_0\_BIT, VK\_IMAGE\_ASPECT\_PLANE\_1\_BIT, or VK\_IMAGE\_ASPECT\_PLANE\_2\_BIT (with VK\_IMAGE\_ASPECT\_PLANE\_2\_BIT valid only for image formats with three planes)

If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE}_3\text{D} \), for each element of \( \text{pRegions} \), imageSubresource\.baseArrayLayer \textbf{must} be 0 and imageSubresource\.layerCount \textbf{must} be 1

If \( \text{dstImage} \) is not a \textit{blocked image}, for each element of \( \text{pRegions} \), bufferRowLength multiplied by the texel block size of \( \text{dstImage} \) \textbf{must} be less than or equal to \( 2^{31} - 1 \)

If \( \text{dstImage} \) is a \textit{blocked image}, for each element of \( \text{pRegions} \), bufferRowLength \textbf{divided by}
the compressed texel block width and then multiplied by the texel block size of \( \text{dstImage} \) must be less than or equal to \( 2^{31} - 1 \)

- VUID-vkCmdCopyBufferToImage-commandBuffer-04052
  If the queue family used to create the \( \text{VkCommandPool} \) which \( \text{commandBuffer} \) was allocated from does not support \( \text{VK_QUEUE_GRAPHICS_BIT} \) or \( \text{VK_QUEUE_COMPUTE_BIT} \), the \( \text{bufferOffset} \) member of any element of \( \text{pRegions} \) must be a multiple of 4.

- VUID-vkCmdCopyBufferToImage-srcImage-04053
  If \( \text{dstImage} \) has a depth/stencil format, the \( \text{bufferOffset} \) member of any element of \( \text{pRegions} \) must be a multiple of 4.

### Valid Usage (Implicit)

- VUID-vkCmdCopyBufferToImage-commandBuffer-parameter
  \( \text{commandBuffer} \) must be a valid \( \text{VkCommandBuffer} \) handle.

- VUID-vkCmdCopyBufferToImage-srcBuffer-parameter
  \( \text{srcBuffer} \) must be a valid \( \text{VkBuffer} \) handle.

- VUID-vkCmdCopyBufferToImage-dstImage-parameter
  \( \text{dstImage} \) must be a valid \( \text{VkImage} \) handle.

- VUID-vkCmdCopyBufferToImage-dstImageLayout-parameter
  \( \text{dstImageLayout} \) must be a valid \( \text{VkImageLayout} \) value.

- VUID-vkCmdCopyBufferToImage-pRegions-parameter
  \( \text{pRegions} \) must be a valid pointer to an array of \( \text{regionCount} \) valid \( \text{VkBufferImageCopy} \) structures.

- VUID-vkCmdCopyBufferToImage-commandBuffer-recording
  \( \text{commandBuffer} \) must be in the recording state.

- VUID-vkCmdCopyBufferToImage-commandBuffer-cmdpool
  The \( \text{VkCommandPool} \) that \( \text{commandBuffer} \) was allocated from must support transfer, graphics, or compute operations.

- VUID-vkCmdCopyBufferToImage-renderpass
  This command must only be called outside of a render pass instance.

- VUID-vkCmdCopyBufferToImage-regionCount-arraylength
  \( \text{regionCount} \) must be greater than 0.

- VUID-vkCmdCopyBufferToImage-commonparent
  Each of \( \text{commandBuffer} \), \( \text{dstImage} \), and \( \text{srcBuffer} \) must have been created, allocated, or retrieved from the same \( \text{VkDevice} \).
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics Compute</td>
<td></td>
</tr>
</tbody>
</table>

To copy data from an image object to a buffer object, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdCopyImageToBuffer(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkBuffer dstBuffer,
    uint32_t regionCount,
    const VkBufferImageCopy* pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the copy.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the specified region of the source image to the specified region of the destination buffer.

If the `VkFormat` of `srcImage` is a **multi-planar image format**, regions of each plane to be a source of a copy must be specified separately using the `pRegions` member of the `VkBufferImageCopy` structure. In this case, the `aspectMask` of `imageSubresource` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT`. For the purposes of `vkCmdCopyBufferToImage`, each plane of a multi-planar image is treated as having the format listed in **Compatible formats of planes of multi-planar formats** for the plane identified by the `aspectMask` of the corresponding subresource. This applies both to `VkFormat` and to coordinates used in the copy.
which correspond to texels in the \textit{plane} rather than how these texels map to coordinates in the image as a whole.
Valid Usage

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-01831**
  If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-01832**
  If `commandBuffer` is an unprotected command buffer, then `dstBuffer` must not be a protected buffer

- **VUID-vkCmdCopyImageToBuffer-commandBuffer-01833**
  If `commandBuffer` is a protected command buffer, then `dstBuffer` must not be an unprotected buffer

- **VUID-vkCmdCopyImageToBuffer-pRegions-00182**
  The image region specified by each element of `pRegions` must be a region that is contained within `srcImage` if the `srcImage`'s `VkFormat` is not a multi-planar format, and must be a region that is contained within the plane being copied if the `srcImage`'s `VkFormat` is a multi-planar format

- **VUID-vkCmdCopyImageToBuffer-pRegions-00183**
  `dstBuffer` must be large enough to contain all buffer locations that are accessed according to `Buffer and Image Addressing`, for each element of `pRegions`

- **VUID-vkCmdCopyImageToBuffer-pRegions-00184**
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory

- **VUID-vkCmdCopyImageToBuffer-srcImage-00186**
  `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag

- **VUID-vkCmdCopyImageToBuffer-srcImage-01998**
  The format features of `srcImage` must contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`

- **VUID-vkCmdCopyImageToBuffer-srcImage-00187**
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdCopyImageToBuffer-dstBuffer-00191**
  `dstBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag

- **VUID-vkCmdCopyImageToBuffer-dstBuffer-00192**
  If `dstBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdCopyImageToBuffer-srcImage-00188**
  `srcImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

- **VUID-vkCmdCopyImageToBuffer-srcImageLayout-00189**
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- **VUID-vkCmdCopyImageToBuffer-srcImageLayout-01397**
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, `VK_IMAGE_LAYOUT_GENERAL`, `VK_IMAGE_LAYOUT_UNDEFINED`, or `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`.

Chapter 20. Copy Commands | 1231
The `imageSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

The `imageSubresource.baseArrayLayer + imageSubresource.layerCount` of each element of `pRegions` must be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

The `imageOffset` and `imageExtent` members of each element of `pRegions` must respect the image transfer granularity requirements of `commandBuffer`'s command pool's queue family, as described in `VkQueueFamilyProperties`.

`srcImage` must not have been created with flags containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.

For each element of `pRegions`, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of `srcImage` where this refers to the width of the plane of the image involved in the copy in the case of a multi-planar format.

For each element of `pRegions`, `imageOffset.y` and `(imageExtent.height + imageOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `imageSubresource` of `srcImage` where this refers to the height of the plane of the image involved in the copy in the case of a multi-planar format.

If `srcImage` does not have either a depth/stencil or a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the format's texel block size.

If `srcImage` has a multi-planar format, then for each element of `pRegions`, `bufferOffset` must be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in Compatible formats of planes of multi-planar formats.

If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `imageOffset.y` must be 0 and `imageExtent.height` must be 1.

For each element of `pRegions`, `imageOffset.z` and `(imageExtent.depth + imageOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `imageSubresource` of `srcImage`.

If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `imageOffset.z` must be 0 and `imageExtent.depth` must be 1.
If `srcImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` must be a multiple of the compressed texel block width

If `srcImage` is a blocked image, for each element of `pRegions`, `bufferImageHeight` must be a multiple of the compressed texel block height

If `srcImage` is a blocked image, for each element of `pRegions`, all members of `imageOffset` must be a multiple of the corresponding dimensions of the compressed texel block

If `srcImage` is a blocked image, for each element of `pRegions`, `bufferOffset` must be a multiple of the compressed texel block size in bytes

If `srcImage` is a blocked image, for each element of `pRegions`, `imageExtent.width` must be a multiple of the compressed texel block width or `(imageExtent.width + imageOffset.x)` must equal the width of the specified `imageSubresource` of `srcImage`

If `srcImage` is a blocked image, for each element of `pRegions`, `imageExtent.height` must be a multiple of the compressed texel block height or `(imageExtent.height + imageOffset.y)` must equal the height of the specified `imageSubresource` of `srcImage`

If `srcImage` is a blocked image, for each element of `pRegions`, `imageExtent.depth` must be a multiple of the compressed texel block depth or `(imageExtent.depth + imageOffset.z)` must equal the depth of the specified `imageSubresource` of `srcImage`

For each element of `pRegions`, `imageSubresource.aspectMask` must specify aspects present in `srcImage`

If `srcImage` has a multi-planar format, then for each element of `pRegions`, `imageSubresource.aspectMask` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT` (with `VK_IMAGE_ASPECT_PLANE_2_BIT` valid only for image formats with three planes)

If `srcImage` is of type `VK_IMAGE_TYPE_3D`, for each element of `pRegions`, `imageSubresource.baseArrayLayer` must be 0 and `imageSubresource.layerCount` must be 1

If `srcImage` is not a blocked image, for each element of `pRegions`, `bufferRowLength` multiplied by the texel block size of `srcImage` must be less than or equal to $2^{31} - 1$

If `srcImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` divided by the compressed texel block width and then multiplied by the texel block size of `srcImage` must be less than or equal to $2^{31} - 1$
If the queue family used to create the VkCommandPool which commandBuffer was allocated from does not support VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT, the bufferOffset member of any element of pRegions must be a multiple of 4

- VUID-vkCmdCopyImageToBuffer-srcImage-04053
  If srcImage has a depth/stencil format, the bufferOffset member of any element of pRegions must be a multiple of 4

### Valid Usage (Implicit)

- VUID-vkCmdCopyImageToBuffer-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyImageToBuffer-srcImage-parameter
  srcImage must be a valid VkImage handle

- VUID-vkCmdCopyImageToBuffer-srcImageLayout-parameter
  srcImageLayout must be a valid VkImageLayout value

- VUID-vkCmdCopyImageToBuffer-dstBuffer-parameter
  dstBuffer must be a valid VkBuffer handle

- VUID-vkCmdCopyImageToBuffer-pRegions-parameter
  pRegions must be a valid pointer to an array of regionCount valid VkBufferImageCopy structures

- VUID-vkCmdCopyImageToBuffer-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdCopyImageToBuffer-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyImageToBuffer-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdCopyImageToBuffer-regionCount-arraylength
  regionCount must be greater than 0

- VUID-vkCmdCopyImageToBuffer-commonparent
  Each of commandBuffer, dstBuffer, and srcImage must have been created, allocated, or retrieved from the same VkDevice

### Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

For both `vkCmdCopyBufferToImage` and `vkCmdCopyImageToBuffer`, each element of `pRegions` is a structure defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBufferImageCopy {
    VkDeviceSize                bufferOffset;
    uint32_t                     bufferRowLength;
    uint32_t                     bufferImageHeight;
    VkImageSubresourceLayers    imageSubresource;
    VkOffset3D                  imageOffset;
    VkExtent3D                  imageExtent;
} VkBufferImageCopy;
```

- `bufferOffset` is the offset in bytes from the start of the buffer object where the image data is copied from or to.

- `bufferRowLength` and `bufferImageHeight` specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the `imageExtent`.

- `imageSubresource` is a `VkImageSubresourceLayers` used to specify the specific image subresources of the image used for the source or destination image data.

- `imageOffset` selects the initial x, y, z offsets in texels of the sub-region of the source or destination image data.

- `imageExtent` is the size in texels of the image to copy in width, height and depth.

When copying to or from a depth or stencil aspect, the data in buffer memory uses a layout that is a (mostly) tightly packed representation of the depth or stencil data. Specifically:

- data copied to or from the stencil aspect of any depth/stencil format is tightly packed with one `VK_FORMAT_S8_UINT` value per texel.

- data copied to or from the depth aspect of a `VK_FORMAT_D16_UNORM` or `VK_FORMAT_D16_UNORM_S8_UINT` format is tightly packed with one `VK_FORMAT_D16_UNORM` value per texel.

- data copied to or from the depth aspect of a `VK_FORMAT_D32_SFLOAT` or `VK_FORMAT_D32_SFLOAT_S8_UINT` format is tightly packed with one `VK_FORMAT_D32_SFLOAT` value per texel.
data copied to or from the depth aspect of a \texttt{VK_FORMAT_X8_D24_UNORM_PACK32} or \texttt{VK_FORMAT_D24_UNORM_S8_UINT} format is packed with one 32-bit word per texel with the D24 value in the LSBs of the word, and undefined values in the eight MSBs.

\begin{itemize}
\item Note
\begin{itemize}
\item To copy both the depth and stencil aspects of a depth/stencil format, two entries in pRegions can be used, where one specifies the depth aspect in imageSubresource, and the other specifies the stencil aspect.
\end{itemize}
\end{itemize}

Because depth or stencil aspect buffer to image copies may require format conversions on some implementations, they are not supported on queues that do not support graphics.

When copying to a depth aspect, and the \texttt{VK_EXT_depth_range_unrestricted} extension is not enabled, the data in buffer memory must be in the range $[0,1]$, or the resulting values are undefined.

Copies are done layer by layer starting with image layer baseArrayLayer member of imageSubresource. layerCount layers are copied from the source image or to the destination image.

For purpose of valid usage statements here and in related copy commands, a blocked image is defined as:

\begin{itemize}
\item an image with a single-plane, "\_422" format, which is treated as a format with a $2 \times 1$ compressed texel block, or
\item a compressed image.
\end{itemize}

\begin{table}
\centering
\begin{tabular}{|c|}
\hline
Valid Usage \hline
\item VUID-VkBufferImageCopy-bufferRowLength-00195 \hline
\texttt{bufferRowLength} must be 0, or greater than or equal to the \texttt{width} member of \texttt{imageExtent} \hline
\item VUID-VkBufferImageCopy-bufferImageHeight-00196 \hline
\texttt{bufferImageHeight} must be 0, or greater than or equal to the \texttt{height} member of \texttt{imageExtent} \hline
\item VUID-VkBufferImageCopy-aspectMask-00212 \hline
The \texttt{aspectMask} member of \texttt{imageSubresource} must only have a single bit set
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|}
\hline
Valid Usage (Implicit) \hline
\item VUID-VkBufferImageCopy-imageSubresource-parameter \hline
\texttt{imageSubresource} must be a valid \texttt{VkImageSubresourceLayers} structure
\end{tabular}
\end{table}

More extensible versions of the commands to copy between buffers and images are defined below.

To copy data from a buffer object to an image object, call:
// Provided by VK_KHR_copy_commands2
void vkCmdCopyBufferToImage2KHR(
  VkCommandBuffer commandBuffer,
  const VkCopyBufferToImageInfo2KHR* pCopyBufferToImageInfo);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyBufferToImageInfo` is a pointer to a `VkCopyBufferToImageInfo2KHR` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyBufferToImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-01828
  If `commandBuffer` is an unprotected command buffer, then `srcBuffer` **must** not be a protected buffer
- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-01829
  If `commandBuffer` is an unprotected command buffer, then `dstImage` **must** not be a protected image
- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-01830
  If `commandBuffer` is a protected command buffer, then `dstImage` **must** not be an unprotected image

### Valid Usage (Implicit)

- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdCopyBufferToImage2KHR-pCopyBufferToImageInfo-parameter
  `pCopyBufferToImageInfo` **must** be a valid pointer to a valid `VkCopyBufferToImageInfo2KHR` structure
- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-recording
  `commandBuffer` **must** be in the recording state
- VUID-vkCmdCopyBufferToImage2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute operations
- VUID-vkCmdCopyBufferToImage2KHR-renderpass
  This command **must** only be called outside of a render pass instance

Chapter 20. Copy Commands | 1237
Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics, Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyBufferToImageInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyBufferToImageInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkBuffer srcBuffer;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkBufferImageCopy2KHR* pRegions;
} VkCopyBufferToImageInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcBuffer` is the source buffer.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the copy.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy2KHR` structures specifying the regions to copy.
Valid Usage

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-04565
  If the image region specified by each element of `pRegions` does not contain `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, it must be a region that is contained within `dstImage` if the `dstImage`'s `VkFormat` is not a multi-planar format, and must be a region that is contained within the plane being copied to if the `dstImage`'s `VkFormat` is a multi-planar format.

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-04554
  If the image region specified by each element of `pRegions` does contain `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, the rotated destination region as described in Buffer and Image Addressing with Rotation must be contained within `dstImage`.

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-04555
  If any element of `pRegions` contains `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, then the `dstImage` must not be a blocked image.

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-04556
  If any element of `pRegions` contains `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, then the `dstImage` must be of type `VK_IMAGE_TYPE_2D` and must not be a multi-planar format.

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-00171
  `srcBuffer` must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of `pRegions`.

- VUID-VkCopyBufferToImageInfo2KHR-pRegions-00173
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory.

- VUID-VkCopyBufferToImageInfo2KHR-srcBuffer-00174
  `srcBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag.

- VUID-VkCopyBufferToImageInfo2KHR-srcBuffer-00176
  If `srcBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-01997
  The `format features` of `dstImage` must contain `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-00177
  `dstImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag.

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-00178
  If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- VUID-VkCopyBufferToImageInfo2KHR-dstImageLayout-00180
  `dstImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`.

- VUID-VkCopyBufferToImageInfo2KHR-dstImageLayout-00180
  `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in...
pRegions at the time this command is executed on a VkDevice

- VUID-VkCopyBufferToImageInfo2KHR-dstImageLayout-01396
dstImageLayout must be VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_GENERAL, or VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR

- VUID-VkCopyBufferToImageInfo2KHR-imageSubresource-01701
  The imageSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

- VUID-VkCopyBufferToImageInfo2KHR-imageSubresource-01702
  The imageSubresource.baseArrayLayer + imageSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

- VUID-VkCopyBufferToImageInfo2KHR-imageOffset-01793
  The imageOffset and imageExtent members of each element of pRegions must respect the image transfer granularity requirements of commandBuffer's command pool's queue family, as described in VkQueueFamilyProperties

- VUID-VkCopyBufferToImageInfo2KHR-dstImage-02543
dstImage must not have been created with flags containing VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT

- VUID-VkCopyBufferToImageInfo2KHR-commandBuffer-04477
  If the queue family used to create the VkCommandPool which commandBuffer was allocated from does not support VK_QUEUE_GRAPHICS_BIT, for each element of pRegions, the aspectMask member of imageSubresource must not be VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT.

- VUID-VkCopyBufferToImageInfo2KHR-imageOffset-00197
  For each element of pRegions not containing VkCopyCommandTransformInfoQCOM in itspNext chain, , imageOffset.x and (imageExtent.width + imageOffset.x) must both be greater than or equal to 0 and less than or equal to the width of the specified imageSubresource of dstImage where this refers to the width of the plane of the image involved in the copy in the case of a multi-planar format

- VUID-VkCopyBufferToImageInfo2KHR-imageOffset-00198
  For each element of pRegions not containing VkCopyCommandTransformInfoQCOM in itspNext chain, , imageOffset.y and (imageExtent.height + imageOffset.y) must both be greater than or equal to 0 and less than or equal to the height of the specified imageSubresource of dstImage where this refers to the height of the plane of the image involved in the copy in the case of a multi-planar format

- VUID-VkCopyBufferToImageInfo2KHR-bufferOffset-01558
  If dstImage does not have either a depth/stencil or a multi-planar format, then for each element of pRegions, bufferOffset must be a multiple of the format's texel block size

- VUID-VkCopyBufferToImageInfo2KHR-bufferOffset-01559
  If dstImage has a multi-planar format, then for each element of pRegions, bufferOffset must be a multiple of the element size of the compatible format for the format and the aspectMask of the imageSubresource as defined in Compatible formats of planes of multi-planar formats
If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `imageOffset.y` must be 0 and `imageExtent.height` must be 1.

For each element of `pRegions`, `imageOffset.z` and `(imageExtent.depth + imageOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `imageSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `imageOffset.z` must be 0 and `imageExtent.depth` must be 1.

If `dstImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` must be a multiple of the compressed texel block width.

If `dstImage` is a blocked image, for each element of `pRegions`, `bufferImageHeight` must be a multiple of the compressed texel block height.

If `dstImage` is a blocked image, for each element of `pRegions`, all members of `imageOffset` must be a multiple of the corresponding dimensions of the compressed texel block.

If `dstImage` is a blocked image, for each element of `pRegions`, `bufferOffset` must be a multiple of the compressed texel block size in bytes.

If `dstImage` is a blocked image, for each element of `pRegions`, `imageExtent.width` must be a multiple of the width of the specified `imageSubresource` of `dstImage`.

If `dstImage` is a blocked image, for each element of `pRegions`, `imageExtent.height` must be a multiple of the compressed texel block height or `(imageExtent.height + imageOffset.y)` must equal the height of the specified `imageSubresource` of `dstImage`.

If `dstImage` is a blocked image, for each element of `pRegions`, `imageExtent.depth` must be a multiple of the compressed texel block depth or `(imageExtent.depth + imageOffset.z)` must equal the depth of the specified `imageSubresource` of `dstImage`.

For each element of `pRegions`, `imageSubresource.aspectMask` must specify aspects present in `dstImage`.

If `dstImage` has a multi-planar format, then for each element of `pRegions`, `imageSubresource.aspectMask` must be `VK_IMAGE_ASPECT_PLANE_0_BIT`, `VK_IMAGE_ASPECT_PLANE_1_BIT`, or `VK_IMAGE_ASPECT_PLANE_2_BIT` (with `VK_IMAGE_ASPECT_PLANE_2_BIT` valid only for image formats with three planes).
If `dstImage` is of type `VK_IMAGE_TYPE_3D`, for each element of `pRegions`, `imageSubresource.baseArrayLayer` must be 0 and `imageSubresource.layerCount` must be 1

- **VUID-VkCopyBufferToImageInfo2KHR-pRegions-04725**
  If `dstImage` is not a blocked image, for each element of `pRegions`, `bufferRowLength` multiplied by the texel block size of `dstImage` must be less than or equal to $2^{31} - 1$

- **VUID-VkCopyBufferToImageInfo2KHR-pRegions-04726**
  If `dstImage` is a blocked image, for each element of `pRegions`, `bufferRowLength` divided by the compressed texel block width and then multiplied by the texel block size of `dstImage` must be less than or equal to $2^{31} - 1$

- **VUID-VkCopyBufferToImageInfo2KHR-commandBuffer-04052**
  If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, the `bufferOffset` member of any element of `pRegions` must be a multiple of 4

- **VUID-VkCopyBufferToImageInfo2KHR-srcImage-04053**
  If `dstImage` has a depth/stencil format, the `bufferOffset` member of any element of `pRegions` must be a multiple of 4

**Valid Usage (Implicit)**

- **VUID-VkCopyBufferToImageInfo2KHR-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2_KHR`

- **VUID-VkCopyBufferToImageInfo2KHR-pNext-pNext**
  `pNext` must be `NULL`

- **VUID-VkCopyBufferToImageInfo2KHR-srcBuffer-parameter**
  `srcBuffer` must be a valid `VkBuffer` handle

- **VUID-VkCopyBufferToImageInfo2KHR-dstImage-parameter**
  `dstImage` must be a valid `VkImage` handle

- **VUID-VkCopyBufferToImageInfo2KHR-dstImageLayout-parameter**
  `dstImageLayout` must be a valid `VkImageLayout` value

- **VUID-VkCopyBufferToImageInfo2KHR-pRegions-parameter**
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy2KHR` structures

- **VUID-VkCopyBufferToImageInfo2KHR-regionCount-arraylength**
  `regionCount` must be greater than 0

- **VUID-VkCopyBufferToImageInfo2KHR-commonparent**
  Both of `dstImage`, and `srcBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

To copy data from an image object to a buffer object, call:
```c
// Provided by VK_KHR_copy_commands2
void vkCmdCopyImageToBuffer2KHR(
    VkCommandBuffer                             commandBuffer,  
    const VkCopyImageToBufferInfo2KHR*          pCopyImageToBufferInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pCopyImageToBufferInfo` is a pointer to a `VkCopyImageToBufferInfo2KHR` structure describing the copy parameters.

This command is functionally identical to `vkCmdCopyImageToBuffer`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

### Valid Usage

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-01831
  If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-01832
  If `commandBuffer` is an unprotected command buffer, then `dstBuffer` must not be a protected buffer

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-01833
  If `commandBuffer` is a protected command buffer, then `dstBuffer` must not be an unprotected buffer

### Valid Usage (Implicit)

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyImageToBuffer2KHR-pCopyImageToBufferInfo-parameter
  `pCopyImageToBufferInfo` must be a valid pointer to a valid `VkCopyImageToBufferInfo2KHR` structure

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyImageToBuffer2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdCopyImageToBuffer2KHR-renderpass
  This command must only be called outside of a render pass instance
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Transfer</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyImageToBufferInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkCopyImageToBufferInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkBuffer dstBuffer;
    uint32_t regionCount;
    const VkBufferImageCopy2KHR* pRegions;
} VkCopyImageToBufferInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the copy.
- `dstBuffer` is the destination buffer.
- `regionCount` is the number of regions to copy.
- `pRegions` is a pointer to an array of `VkBufferImageCopy2KHR` structures specifying the regions to copy.
Valid Usage

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-04566**
  If the image region specified by each element of `pRegions` does not contain `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, it **must** be a region that is contained within `srcImage` if the `srcImage`'s `VkFormat` is not a multi-planar format, and **must** be a region that is contained within the plane being copied if the `srcImage`'s `VkFormat` is a multi-planar format.

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-04557**
  If the image region specified by each element of `pRegions` does contain `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, the rotated source region as described in Buffer and Image Addressing with Rotation **must** be contained within `srcImage`.

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-04558**
  If any element of `pRegions` contains `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, then the `srcImage` **must** not be a blocked image.

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-04559**
  If any element of `pRegions` contains `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, then the `srcImage` **must** be of type `VK_IMAGE_TYPE_2D`, and **must** not be a multi-planar format.

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-00183**
  `dstBuffer` **must** be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of `pRegions`.

- **VUID-VkCopyImageToBufferInfo2KHR-pRegions-00184**
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory.

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00186**
  `srcImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag.

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-01998**
  The format features of `srcImage` **must** contain `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`.

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00187**
  If `srcImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-VkCopyImageToBufferInfo2KHR-dstBuffer-00191**
  `dstBuffer` **must** have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

- **VUID-VkCopyImageToBufferInfo2KHR-dstBuffer-00192**
  If `dstBuffer` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00188**
  `srcImage` **must** have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`.

- **VUID-VkCopyImageToBufferInfo2KHR-srcImageLayout-00189**
  `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in...
**pRegions** at the time this command is executed on a **VkDevice**

- **VUID-VkCopyImageToBufferInfo2KHR-srcImageLayout-01397**
  
  `srcImageLayout` **must** be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL**, **VK_IMAGE_LAYOUT_GENERAL**, or **VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR**

- **VUID-VkCopyImageToBufferInfo2KHR-imageSubresource-01703**
  
  The `imageSubresource.mipLevel` member of each element of **pRegions** **must** be less than the `mipLevels` specified in **VkImageCreateInfo** when **srcImage** was created

- **VUID-VkCopyImageToBufferInfo2KHR-imageSubresource-01704**
  
  The `imageSubresource.baseArrayLayer + imageSubresource.layerCount` of each element of **pRegions** **must** be less than or equal to the `arrayLayers` specified in **VkImageCreateInfo** when **srcImage** was created

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-01794**
  
  The `imageOffset` and `imageExtent` members of each element of **pRegions** **must** respect the image transfer granularity requirements of **commandBuffer**’s command pool’s queue family, as described in **VkQueueFamilyProperties**

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-02544**
  
  **srcImage** **must** not have been created with flags containing **VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT**

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-00197**
  
  For each element of **pRegions** not containing **VkCopyCommandTransformInfoQCOM** in its `pNext` chain, `imageOffset.x` and `(imageExtent.width + imageOffset.x)` **must** both be greater than or equal to 0 and less than or equal to the width of the specified `imageSubresource` of **srcImage** where this refers to the width of the `plane` of the image involved in the copy in the case of a **multi-planar format**

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-00198**
  
  For each element of **pRegions** not containing **VkCopyCommandTransformInfoQCOM** in its `pNext` chain, `imageOffset.y` and `(imageExtent.height + imageOffset.y)` **must** both be greater than or equal to 0 and less than or equal to the height of the specified `imageSubresource` of **srcImage** where this refers to the height of the `plane` of the image involved in the copy in the case of a **multi-planar format**

- **VUID-VkCopyImageToBufferInfo2KHR-bufferOffset-01558**
  
  If **srcImage** does not have either a depth/stencil or a **multi-planar format**, then for each element of **pRegions**, `bufferOffset` **must** be a multiple of the format’s texel block size

- **VUID-VkCopyImageToBufferInfo2KHR-bufferOffset-01559**
  
  If **srcImage** has a **multi-planar format**, then for each element of **pRegions**, `bufferOffset` **must** be a multiple of the element size of the compatible format for the format and the `aspectMask` of the `imageSubresource` as defined in **Compatible formats of planes of multi-planar formats**

- **VUID-VkCopyImageToBufferInfo2KHR-srcImage-00199**
  
  If **srcImage** is of type **VK_IMAGE_TYPE_1D**, then for each element of **pRegions**, `imageOffset.y` **must** be 0 and `imageExtent.height` **must** be 1

- **VUID-VkCopyImageToBufferInfo2KHR-imageOffset-00200**
  
  For each element of **pRegions**, `imageOffset.z` and `(imageExtent.depth + imageOffset.z)` **must**
both be greater than or equal to 0 and less than or equal to the depth of the specified imageSubresource of srcImage

• VUID-VkCopyImageToBufferInfo2KHR-srcImage-00201
  If srcImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, imageOffset.z must be 0 and imageExtent.depth must be 1

• VUID-VkCopyImageToBufferInfo2KHR-bufferRowLength-00203
  If srcImage is a blocked image, for each element of pRegions, bufferRowLength must be a multiple of the compressed texel block width

• VUID-VkCopyImageToBufferInfo2KHR-bufferImageHeight-00204
  If srcImage is a blocked image, for each element of pRegions, bufferImageHeight must be a multiple of the compressed texel block height

• VUID-VkCopyImageToBufferInfo2KHR-imageOffset-00205
  If srcImage is a blocked image, for each element of pRegions, all members of imageOffset must be a multiple of the corresponding dimensions of the compressed texel block

• VUID-VkCopyImageToBufferInfo2KHR-bufferOffset-00206
  If srcImage is a blocked image, for each element of pRegions, bufferOffset must be a multiple of the compressed texel block size in bytes

• VUID-VkCopyImageToBufferInfo2KHR-imageExtent-00207
  If srcImage is a blocked image, for each element of pRegions, imageExtent.width must be a multiple of the compressed texel block width or (imageExtent.width + imageOffset.x) must equal the width of the specified imageSubresource of srcImage

• VUID-VkCopyImageToBufferInfo2KHR-imageExtent-00208
  If srcImage is a blocked image, for each element of pRegions, imageExtent.height must be a multiple of the compressed texel block height or (imageExtent.height + imageOffset.y) must equal the height of the specified imageSubresource of srcImage

• VUID-VkCopyImageToBufferInfo2KHR-imageExtent-00209
  If srcImage is a blocked image, for each element of pRegions, imageExtent.depth must be a multiple of the compressed texel block depth or (imageExtent.depth + imageOffset.z) must equal the depth of the specified imageSubresource of srcImage

• VUID-VkCopyImageToBufferInfo2KHR-aspectMask-00211
  For each element of pRegions, imageSubresource.aspectMask must specify aspects present in srcImage

• VUID-VkCopyImageToBufferInfo2KHR-aspectMask-01560
  If srcImage has a multi-planar format, then for each element of pRegions, imageSubresource.aspectMask must be VK_IMAGE_ASPECT_PLANE_0_BIT, VK_IMAGE_ASPECT_PLANE_1_BIT, or VK_IMAGE_ASPECT_PLANE_2_BIT (with VK_IMAGE_ASPECT_PLANE_2_BIT valid only for image formats with three planes)

• VUID-VkCopyImageToBufferInfo2KHR-baseArrayLayer-00213
  If srcImage is of type VK_IMAGE_TYPE_3D, for each element of pRegions, imageSubresource.baseArrayLayer must be 0 and imageSubresource.layerCount must be 1

• VUID-VkCopyImageToBufferInfo2KHR-pRegions-04725
  If srcImage is not a blocked image, for each element of pRegions, bufferRowLength multiplied by the texel block size of srcImage must be less than or equal to 2^31-1
If `srcImage` is a **blocked image**, for each element of `pRegions`, `bufferRowLength` divided by the compressed texel block width and then multiplied by the texel block size of `srcImage` **must** be less than or equal to $2^{31}-1$.

If the queue family used to create the `VkCommandPool` which `commandBuffer` was allocated from does not support `VK_QUEUE_GRAPHICS_BIT` or `VK_QUEUE_COMPUTE_BIT`, the `bufferOffset` member of any element of `pRegions` **must** be a multiple of 4.

If `srcImage` has a depth/stencil format, the `bufferOffset` member of any element of `pRegions` **must** be a multiple of 4.

---

**Valid Usage (Implicit)**

- **sType** **must** be `VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2_KHR`
- `pNext` **must** be `NULL`
- `srcImage` **must** be a valid `VkImage` handle
- `srcImageLayout` **must** be a valid `VkImageLayout` value
- `dstBuffer` **must** be a valid `VkBuffer` handle
- `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkBufferImageCopy2KHR` structures
- `regionCount` **must** be greater than 0
- Both of `dstBuffer`, and `srcImage` **must** have been created, allocated, or retrieved from the same `VkDevice`

For both `vkCmdCopyBufferToImage2KHR` and `vkCmdCopyImageToBuffer2KHR`, each element of `pRegions` is a structure defined as:
// Provided by VK_KHR_copy_commands2

typedef struct VkBufferImageCopy2KHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceSize bufferOffset;
    uint32_t bufferRowLength;
    uint32_t bufferImageHeight;
    VkImageSubresourceLayers imageSubresource;
    VkOffset3D imageOffset;
    VkExtent3D imageExtent;
} VkBufferImageCopy2KHR;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **bufferOffset** is the offset in bytes from the start of the buffer object where the image data is copied from or to.

• **bufferRowLength** and **bufferImageHeight** specify in texels a subregion of a larger two- or three-dimensional image in buffer memory, and control the addressing calculations. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the **imageExtent**.

• **imageSubresource** is a **VkImageSubresourceLayers** used to specify the specific image subresources of the image used for the source or destination image data.

• **imageOffset** selects the initial x, y, z offsets in texels of the sub-region of the source or destination image data.

• **imageExtent** is the size in texels of the image to copy in width, height and depth.

This structure is functionally identical to **VkBufferImageCopy** but adds **sType** and **pNext** parameters, allowing it to be more easily extended.

**Valid Usage**

- **VUID-VkBufferImageCopy2KHR-bufferRowLength-00195**
  
  **bufferRowLength** must be 0, or greater than or equal to the width member of **imageExtent**

- **VUID-VkBufferImageCopy2KHR-bufferImageHeight-00196**
  
  **bufferImageHeight** must be 0, or greater than or equal to the height member of **imageExtent**

- **VUID-VkBufferImageCopy2KHR-aspectMask-00212**
  
  The **aspectMask** member of **imageSubresource** must only have a single bit set
Valid Usage (Implicit)

- VUID-VkBufferImageCopy2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2_KHR

- VUID-VkBufferImageCopy2KHR-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of VkCopyCommandTransformInfoQCOM

- VUID-VkBufferImageCopy2KHR-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkBufferImageCopy2KHR-imageSubresource-parameter
  imageSubresource must be a valid VkImageSubresourceLayers structure

For both vkCmdCopyBufferToImage2KHR and vkCmdCopyImageToBuffer2KHR, each region copied can include a rotation. To specify a region with rotation, add the VkCopyCommandTransformInfoQCOM to the pNext chain of VkBufferImageCopy2KHR. When a rotation is specified, Buffer and Image Addressing with Rotation specifies how coordinates of texels in the source region are rotated by transform to produce texel coordinates in the destination region. When rotation is specified, the source or destination image must be a 2D image and must not be blocked image or multi-planar format.

The VkRenderPassTransformBeginInfoQCOM structure is defined as:

```c
// Provided by VK_QCOM_rotated_copy_commands
typedef struct VkCopyCommandTransformInfoQCOM {
    VkStructureType sType;
    const void* pNext;
    VkSurfaceTransformFlagBitsKHR transform;
} VkCopyCommandTransformInfoQCOM;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- transform is a VkSurfaceTransformFlagBitsKHR value describing the transform to be applied.

Valid Usage

- VUID-VkCopyCommandTransformInfoQCOM-transform-04560
  transform must be VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR,
  VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR,
  VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR,
  VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR
20.4.1. Buffer and Image Addressing

Pseudocode for image/buffer addressing of uncompressed formats is:

```c
rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

texelBlockSize = <texel block size of the format of the src/dstImage>;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x) * texelBlockSize;

where x,y,z range from (0,0,0) to region->imageExtent.{width,height,depth}.
```

Note that `imageOffset` does not affect addressing calculations for buffer memory. Instead, `bufferOffset` can be used to select the starting address in buffer memory.

For block-compressed formats, all parameters are still specified in texels rather than compressed texel blocks, but the addressing math operates on whole compressed texel blocks. Pseudocode for compressed copy addressing is:
rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

compressedTexelBlockSizeInBytes = <compressed texel block size taken from the src /dstImage>;
rowLength = (rowLength + compressedTexelBlockWidth - 1) / compressedTexelBlockWidth;
imageHeight = (imageHeight + compressedTexelBlockHeight - 1) / compressedTexelBlockHeight;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x) * compressedTexelBlockSizeInBytes;

where x,y,z range from (0,0,0) to region->imageExtent.{width/compressedTexelBlockWidth,height/compressedTexelBlockHeight,depth/compressedTexelBlockDepth}.

Copying to or from block-compressed images is typically done in multiples of the compressed texel block size. For this reason the imageExtent must be a multiple of the compressed texel block dimension. There is one exception to this rule which is required to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions:

- If imageExtent.width is not a multiple of the compressed texel block width, then (imageExtent.width + imageOffset.x) must equal the image subresource width.
- If imageExtent.height is not a multiple of the compressed texel block height, then (imageExtent.height + imageOffset.y) must equal the image subresource height.
- If imageExtent.depth is not a multiple of the compressed texel block depth, then (imageExtent.depth + imageOffset.z) must equal the image subresource depth.

This allows the last compressed texel block of the image in each non-multiple dimension to be included as a source or destination of the copy.

20.4.2. Buffer and Image Addressing with Rotation

When VkCopyCommandTransformInfoQCOM is in the pNext chain of VkBufferImageCopy2KHR, a rotated copy is specified. For both vkCmdCopyImageToBuffer2KHR and vkCmdCopyBufferToImage2KHR, a rotation is applied to the region used for image accesses, but a non-rotated region is used for buffer accesses. In the case of rotated vkCmdCopyImageToBuffer2KHR, the source image region is rotated. In the case of rotated vkCmdCopyBufferToImage2KHR, the destination image region is rotated.

For a rotated copy, The following description of rotated addressing replaces the description in Buffer and Image Addressing.
The following code computes rotation of unnormalized coordinates.

```c
// Forward rotation of unnormalized coordinates
VkOffset2D RotateUV(VkOffset2D in, VkSurfaceTransformFlagBitsKHR flags)
{
    VkOffset2D output;
    switch (flags)
    {
        case VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR:
            out.x = in.x;
            out.y = in.y;
            break;
        case VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR:
            out.x = -in.y;
            out.y = in.x;
            break;
        case VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR:
            out.x = -in.x;
            out.y = -in.y;
            break;
        case VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR:
            out.x = in.y;
            out.y = -in.x;
            break;
    }
    return out;
}
```

Pseudocode for image/buffer addressing of uncompressed formats with rotation is:
rowLength = region->bufferRowLength;
if (rowLength == 0)
    rowLength = region->imageExtent.width;

imageHeight = region->bufferImageHeight;
if (imageHeight == 0)
    imageHeight = region->imageExtent.height;

texelBlockSize = <texel block size of the format of the src/dstImage>;

// Buffer addressing is unaffected by rotation:
address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x) * texelBlockSize;

// When copying from buffer to image, the source buffer coordinates x,y,z range from (0,0,0) to
// region->imageExtent.{width,height,depth}. The source extent is rotated by the specified
// VK_SURFACE_TRANSFORM, centered on the imageOffset, to define a rotated destination region.
// For each source buffer texel with coordinates (x,y) the rotated destination image texel has
// coordinates (x',y') defined as:
// (x',y')= RotateUV(x,y) + ImageOffset.{x,y}

// When copying from image to buffer, the destination buffer coordinates x,y,z
range from (0,0,0) to
// region->imageExtent.{width,height,depth}. The destination extent is rotated by the specified
// VK_SURFACE_TRANSFORM, centered on the imageOffset, to define a rotated source region. For each destination
// buffer texel with coordinates (x,y) the rotated source image texel has coordinates
// (x',y') defined as:
// (x',y')= RotateUV(x,y) + ImageOffset.{x,y}

Note that imageOffset does not affect addressing calculations for buffer memory. Instead, bufferOffset can be used to select the starting address in buffer memory.

20.5. Image Copies with Scaling

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:
void vkCmdBlitImage(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkImage dstImage,
    VkImageLayout dstImageLayout,
    uint32_t regionCount,
    const VkImageBlit* pRegions,
    VkFilter filter);

• **commandBuffer** is the command buffer into which the command will be recorded.
• **srcImage** is the source image.
• **srcImageLayout** is the layout of the source image subresources for the blit.
• **dstImage** is the destination image.
• **dstImageLayout** is the layout of the destination image subresources for the blit.
• **regionCount** is the number of regions to blit.
• **pRegions** is a pointer to an array of **VkImageBlit** structures specifying the regions to blit.
• **filter** is a **VkFilter** specifying the filter to apply if the blits require scaling.

vkCmdBlitImage must not be used for multisampled source or destination images. Use vkCmdResolveImage for this purpose.

As the sizes of the source and destination extents can differ in any dimension, texels in the source extent are scaled and filtered to the destination extent. Scaling occurs via the following operations:

- For each destination texel, the integer coordinate of that texel is converted to an unnormalized texture coordinate, using the effective inverse of the equations described in **unnormalized to integer conversion**:

  \[
  u_{\text{base}} = i + \frac{1}{2} \\
  v_{\text{base}} = j + \frac{1}{2} \\
  w_{\text{base}} = k + \frac{1}{2}
  \]

- These base coordinates are then offset by the first destination offset:

  \[
  u_{\text{offset}} = u_{\text{base}} - x_{\text{dst0}} \\
  v_{\text{offset}} = v_{\text{base}} - y_{\text{dst0}}
  \]
\[ w_{\text{offset}} = w_{\text{base}} - z_{\text{dst0}} \]

\[ a_{\text{offset}} = a - \text{baseArrayCount}_{\text{dst}} \]

- The scale is determined from the source and destination regions, and applied to the offset coordinates:

\[ \text{scale}_u = \frac{(x_{\text{src1}} - x_{\text{src0}})}{(x_{\text{dst1}} - x_{\text{dst0}})} \]

\[ \text{scale}_v = \frac{(y_{\text{src1}} - y_{\text{src0}})}{(y_{\text{dst1}} - y_{\text{dst0}})} \]

\[ \text{scale}_w = \frac{(z_{\text{src1}} - z_{\text{src0}})}{(z_{\text{dst1}} - z_{\text{dst0}})} \]

\[ u_{\text{scaled}} = u_{\text{offset}} \times \text{scale}_u \]

\[ v_{\text{scaled}} = v_{\text{offset}} \times \text{scale}_v \]

\[ w_{\text{scaled}} = w_{\text{offset}} \times \text{scale}_w \]

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from \textit{srcImage}:

\[ u = u_{\text{scaled}} + x_{\text{src0}} \]

\[ v = v_{\text{scaled}} + y_{\text{src0}} \]

\[ w = w_{\text{scaled}} + z_{\text{src0}} \]

\[ q = \text{mipLevel} \]

\[ a = a_{\text{offset}} + \text{baseArrayCount}_{\text{src}} \]

These coordinates are used to sample from the source image, as described in \textit{Image Operations chapter}, with the filter mode equal to that of \textit{filter}, a mipmap mode of \texttt{VK_SAMPLER_MIPMAP_MODE_NEAREST} and an address mode of \texttt{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE}. 

1256 | Chapter 20. Copy Commands
Implementations must clamp at the edge of the source image, and may additionally clamp to the edge of the source region.

**Note**

Due to allowable rounding errors in the generation of the source texture coordinates, it is not always possible to guarantee exactly which source texels will be sampled for a given blit. As rounding errors are implementation dependent, the exact results of a blitting operation are also implementation dependent.

Blits are done layer by layer starting with the baseArrayLayer member of srcSubresource for the source and dstSubresource for the destination. layerCount layers are blitted to the destination image.

When blitting 3D textures, slices in the destination region bounded by dstOffsets[0].z and dstOffsets[1].z are sampled from slices in the source region bounded by srcOffsets[0].z and srcOffsets[1].z. If the filter parameter is VK_FILTER_LINEAR then the value sampled from the source image is taken by doing linear filtering using the interpolated z coordinate represented by w in the previous equations. If the filter parameter is VK_FILTER_NEAREST then the value sampled from the source image is taken from the single nearest slice, with an implementation-dependent arithmetic rounding mode.

The following filtering and conversion rules apply:

- Integer formats can only be converted to other integer formats with the same signedness.
- No format conversion is supported between depth/stencil images. The formats must match.
- Format conversions on unorm, snorm, unscaled and packed float formats of the copied aspect of the image are performed by first converting the pixels to float values.
- For sRGB source formats, nonlinear RGB values are converted to linear representation prior to filtering.
- After filtering, the float values are first clamped and then cast to the destination image format. In case of sRGB destination format, linear RGB values are converted to nonlinear representation before writing the pixel to the image.

Signed and unsigned integers are converted by first clamping to the representable range of the destination format, then casting the value.
Valid Usage

- VUID-vkCmdBlitImage-commandBuffer-01834
  If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image.

- VUID-vkCmdBlitImage-commandBuffer-01835
  If `commandBuffer` is an unprotected command buffer, then `dstImage` must not be a protected image.

- VUID-vkCmdBlitImage-commandBuffer-01836
  If `commandBuffer` is a protected command buffer, then `dstImage` must not be an unprotected image.

- VUID-vkCmdBlitImage-pRegions-00215
  The source region specified by each element of `pRegions` must be a region that is contained within `srcImage`.

- VUID-vkCmdBlitImage-pRegions-00216
  The destination region specified by each element of `pRegions` must be a region that is contained within `dstImage`.

- VUID-vkCmdBlitImage-pRegions-00217
  The union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory with any texel that may be sampled during the blit operation.

- VUID-vkCmdBlitImage-srcImage-01999
  The `format features` of `srcImage` must contain `VK_FORMAT_FEATURE_BLIT_SRC_BIT`.

- VUID-vkCmdBlitImage-srcImage-01561
  `srcImage` must not use a format listed in Formats requiring sampler Y’C_bC_r conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views.

- VUID-vkCmdBlitImage-srcImage-00219
  `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag.

- VUID-vkCmdBlitImage-srcImage-00220
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

- VUID-vkCmdBlitImage-srcImageLayout-00221
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.

- VUID-vkCmdBlitImage-srcImageLayout-01398
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.

- VUID-vkCmdBlitImage-dstImage-02000
  The `format features` of `dstImage` must contain `VK_FORMAT_FEATURE_BLIT_DST_BIT`.

- VUID-vkCmdBlitImage-dstImage-01562
  `dstImage` must not use a format listed in Formats requiring sampler Y’C_bC_r conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views.
• VUID-vkCmdBlitImage-dstImage-00224
dstImage must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT usage flag

• VUID-vkCmdBlitImage-dstImage-00225
If dstImage is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• VUID-vkCmdBlitImage-dstImageLayout-00226
dstImageLayout must specify the layout of the image subresources of dstImage specified in pRegions at the time this command is executed on a VkDevice

• VUID-vkCmdBlitImage-dstImageLayout-01399
dstImageLayout must be VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL

• VUID-vkCmdBlitImage-srcImage-00229
If either of srcImage or dstImage was created with a signed integer VkFormat, the other must also have been created with a signed integer VkFormat

• VUID-vkCmdBlitImage-srcImage-00230
If either of srcImage or dstImage was created with an unsigned integer VkFormat, the other must also have been created with an unsigned integer VkFormat

• VUID-vkCmdBlitImage-srcImage-00231
If either of srcImage or dstImage was created with a depth/stencil format, the other must have exactly the same format

• VUID-vkCmdBlitImage-srcImage-00232
If srcImage was created with a depth/stencil format, filter must be VK_FILTER_NEAREST

• VUID-vkCmdBlitImage-srcImage-00233
srcImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT

• VUID-vkCmdBlitImage-dstImage-00234
dstImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT

• VUID-vkCmdBlitImage-filter-02001
If filter is VK_FILTER_LINEAR, then the format features of srcImage must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

• VUID-vkCmdBlitImage-filter-02002
If filter is VK_FILTER_CUBIC_EXT, then the format features of srcImage must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

• VUID-vkCmdBlitImage-filter-00237
If filter is VK_FILTER_CUBIC_EXT, srcImage must be of type VK_IMAGE_TYPE_2D

• VUID-vkCmdBlitImage-srcSubresource-01705
The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created

• VUID-vkCmdBlitImage-dstSubresource-01706
The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

• VUID-vkCmdBlitImage-srcSubresource-01707
The srcSubresource.baseArrayLayer + srcSubresource.layerCount of each element of
pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created

- VUID-vkCmdBlitImage-dstSubresource-01708
  The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

- VUID-vkCmdBlitImage-dstImage-02545
dstImage and srcImage must not have been created with flags containing VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT

- VUID-vkCmdBlitImage-srcImage-00240
  If either srcImage or dstImage is of type VK_IMAGE_TYPE_3D, then for each element of pRegions, srcSubresource.baseArrayLayer and dstSubresource.baseArrayLayer must each be 0, and srcSubresource.layerCount and dstSubresource.layerCount must each be 1.

- VUID-vkCmdBlitImage-aspectMask-00241
  For each element of pRegions, srcSubresource.aspectMask must specify aspects present in srcImage

- VUID-vkCmdBlitImage-aspectMask-00242
  For each element of pRegions, dstSubresource.aspectMask must specify aspects present in dstImage

- VUID-vkCmdBlitImage-srcOffset-00243
  For each element of pRegions, srcOffsets[0].x and srcOffsets[1].x must both be greater than or equal to 0 and less than or equal to the width of the specified srcSubresource of srcImage

- VUID-vkCmdBlitImage-srcOffset-00244
  For each element of pRegions, srcOffsets[0].y and srcOffsets[1].y must both be greater than or equal to 0 and less than or equal to the height of the specified srcSubresource of srcImage

- VUID-vkCmdBlitImage-srcImage-00245
  If srcImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, srcOffsets[0].y must be 0 and srcOffsets[1].y must be 1

- VUID-vkCmdBlitImage-srcImage-00246
  For each element of pRegions, srcOffsets[0].z and srcOffsets[1].z must both be greater than or equal to 0 and less than or equal to the depth of the specified srcSubresource of srcImage

- VUID-vkCmdBlitImage-dstOffset-00247
  If srcImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, srcOffsets[0].z must be 0 and srcOffsets[1].z must be 1

- VUID-vkCmdBlitImage-dstImage-00248
  For each element of pRegions, dstOffsets[0].x and dstOffsets[1].x must both be greater than or equal to 0 and less than or equal to the width of the specified dstSubresource of dstImage

- VUID-vkCmdBlitImage-dstImage-00249
  For each element of pRegions, dstOffsets[0].y and dstOffsets[1].y must both be greater than or equal to 0 and less than or equal to the height of the specified dstSubresource of dstImage
If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffsets[0].y` must be 0 and `dstOffsets[1].y` must be 1.

For each element of `pRegions`, `dstOffsets[0].z` and `dstOffsets[1].z` must both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffsets[0].z` must be 0 and `dstOffsets[1].z` must be 1.

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `srcImage` must be a valid `VkImage` handle
- `srcImageLayout` must be a valid `VkImageLayout` value
- `dstImage` must be a valid `VkImage` handle
- `dstImageLayout` must be a valid `VkImageLayout` value
- `pRegions` must be a valid pointer to an array of `regionCount` valid `VkImageBlit` structures
- `filter` must be a valid `VkFilter` value
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- `regionCount` must be greater than 0
- Each of `commandBuffer`, `dstImage`, and `srcImage` must have been created, allocated, or retrieved from the same `VkDevice`
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkImageBlit` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageBlit {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D               srcOffsets[2];
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D               dstOffsets[2];
} VkImageBlit;
```

- `srcSubresource` is the subresource to blit from.
- `srcOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the destination region within `dstSubresource`.

For each element of the `pRegions` array, a blit operation is performed for the specified source and destination regions.

Valid Usage

- VUID-VkImageBlit-aspectMask-00238
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must match.
- VUID-VkImageBlit-layerCount-00239
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match.
Valid Usage (Implicit)

- VUID-VkImageBlit-srcSubresource-parameter
  
  `srcSubresource` **must** be a valid `VkImageSubresourceLayers` structure

- VUID-VkImageBlit-dstSubresource-parameter
  
  `dstSubresource` **must** be a valid `VkImageSubresourceLayers` structure

A more extensible version of the blit image command is defined below.

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdBlitImage2KHR(
    VkCommandBuffer commandBuffer,
    const VkBlitImageInfo2KHR* pBlitImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pBlitImageInfo` is a pointer to a `VkBlitImageInfo2KHR` structure describing the blit parameters.

This command is functionally identical to `vkCmdBlitImage`, but includes extensible sub-structures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

Valid Usage

- VUID-vkCmdBlitImage2KHR-commandBuffer-01834
  
  If `commandBuffer` is an unprotected command buffer, then `srcImage` **must** not be a protected image

- VUID-vkCmdBlitImage2KHR-commandBuffer-01835
  
  If `commandBuffer` is an unprotected command buffer, then `dstImage` **must** not be a protected image

- VUID-vkCmdBlitImage2KHR-commandBuffer-01836
  
  If `commandBuffer` is a protected command buffer, then `dstImage` **must** not be an unprotected image
Valid Usage (Implicit)

- VUID-vkCmdBlitImage2KHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdBlitImage2KHR-pBlitImageInfo-parameter
  `pBlitImageInfo` must be a valid pointer to a valid `VkBlitImageInfo2KHR` structure
- VUID-vkCmdBlitImage2KHR-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdBlitImage2KHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- VUID-vkCmdBlitImage2KHR-renderpass
  This command must only be called outside of a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkBlitImageInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkBlitImageInfo2KHR {
    VkStructureType           sType;
    const void*               pNext;
    VkImage                   srcImage;
    VkImageLayout             srcImageLayout;
    VkImage                   dstImage;
    VkImageLayout             dstImageLayout;
    uint32_t                  regionCount;
    const VkImageBlit2KHR*    pRegions;
    VkFilter                  filter;
} VkBlitImageInfo2KHR;
```
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **srcImage** is the source image.
• **srcImageLayout** is the layout of the source image subresources for the blit.
• **dstImage** is the destination image.
• **dstImageLayout** is the layout of the destination image subresources for the blit.
• **regionCount** is the number of regions to blit.
• **pRegions** is a pointer to an array of **VkImageBlit2KHR** structures specifying the regions to blit.
• **filter** is a **VkFilter** specifying the filter to apply if the blits require scaling.
Valid Usage

- VUID-VkBlitImageInfo2KHR-pRegions-00215
  The source region specified by each element of `pRegions` must be a region that is contained within `srcImage`

- VUID-VkBlitImageInfo2KHR-pRegions-00216
  The destination region specified by each element of `pRegions` must be a region that is contained within `dstImage`

- VUID-VkBlitImageInfo2KHR-pRegions-00217
  The union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory with any texel that may be sampled during the blit operation

- VUID-VkBlitImageInfo2KHR-srcImage-01999
  The format features of `srcImage` must contain `VK_FORMAT_FEATURE_BLIT_SRC_BIT`

- VUID-VkBlitImageInfo2KHR-srcImage-01561
  `srcImage` must not use a format listed in Formats requiring sampler YC_bC_r conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views

- VUID-VkBlitImageInfo2KHR-srcImage-00219
  `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag

- VUID-VkBlitImageInfo2KHR-srcImage-00220
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkBlitImageInfo2KHR-srcImageLayout-00221
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-VkBlitImageInfo2KHR-srcImageLayout-01398
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

- VUID-VkBlitImageInfo2KHR-dstImage-02000
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_BLIT_DST_BIT`

- VUID-VkBlitImageInfo2KHR-dstImage-01562
  `dstImage` must not use a format listed in Formats requiring sampler YC_bC_r conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views

- VUID-VkBlitImageInfo2KHR-dstImage-00224
  `dstImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag

- VUID-VkBlitImageInfo2KHR-dstImage-00225
  If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkBlitImageInfo2KHR-dstImageLayout-00226
  `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-VkBlitImageInfo2KHR-dstImageLayout-01399
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL

- VUID-VkBlitImageInfo2KHR-srcImage-00229
  If either of srcImage or dstImage was created with a signed integer VkFormat, the other must also have been created with a signed integer VkFormat

- VUID-VkBlitImageInfo2KHR-srcImage-00230
  If either of srcImage or dstImage was created with an unsigned integer VkFormat, the other must also have been created with an unsigned integer VkFormat

- VUID-VkBlitImageInfo2KHR-srcImage-00231
  If either of srcImage or dstImage was created with a depth/stencil format, the other must have exactly the same format

- VUID-VkBlitImageInfo2KHR-srcImage-00232
  If srcImage was created with a depth/stencil format, filter must be VK_FILTER_NEAREST

- VUID-VkBlitImageInfo2KHR-srcImage-00233
  srcImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT

- VUID-VkBlitImageInfo2KHR-dstImage-00234
  dstImage must have been created with a samples value of VK_SAMPLE_COUNT_1_BIT

- VUID-VkBlitImageInfo2KHR-filter-02001
  If filter is VK_FILTER_LINEAR, then the format features of srcImage must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-VkBlitImageInfo2KHR-filter-02002
  If filter is VK_FILTER_CUBIC_EXT, then the format features of srcImage must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

- VUID-VkBlitImageInfo2KHR-filter-00237
  If filter is VK_FILTER_CUBIC_EXT, srcImage must be of type VK_IMAGE_TYPE_2D

- VUID-VkBlitImageInfo2KHR-srcSubresource-01705
  The srcSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when srcImage was created

- VUID-VkBlitImageInfo2KHR-dstSubresource-01706
  The dstSubresource.mipLevel member of each element of pRegions must be less than the mipLevels specified in VkImageCreateInfo when dstImage was created

- VUID-VkBlitImageInfo2KHR-srcSubresource-01707
  The srcSubresource.baseArrayLayer + srcSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when srcImage was created

- VUID-VkBlitImageInfo2KHR-dstSubresource-01708
  The dstSubresource.baseArrayLayer + dstSubresource.layerCount of each element of pRegions must be less than or equal to the arrayLayers specified in VkImageCreateInfo when dstImage was created

- VUID-VkBlitImageInfo2KHR-dstImage-02545
  dstImage and srcImage must not have been created with flags containing VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT

- VUID-VkBlitImageInfo2KHR-srcImage-00240
  If either srcImage or dstImage is of type VK_IMAGE_TYPE_3D, then for each element of
pRegions, srcSubresource.baseArrayLayer and dstSubresource.baseArrayLayer must each be 0, and srcSubresource.layerCount and dstSubresource.layerCount must each be 1.

- VUID-VkBlitImageInfo2KHR-aspectMask-00241
  For each element of pRegions, srcSubresource.aspectMask must specify aspects present in srcImage

- VUID-VkBlitImageInfo2KHR-aspectMask-00242
  For each element of pRegions, dstSubresource.aspectMask must specify aspects present in dstImage

- VUID-VkBlitImageInfo2KHR-srcOffset-00243
  For each element of pRegions, srcOffsets[0].x and srcOffsets[1].x must both be greater than or equal to 0 and less than or equal to the width of the specified srcSubresource of srcImage

- VUID-VkBlitImageInfo2KHR-srcOffset-00244
  For each element of pRegions, srcOffsets[0].y and srcOffsets[1].y must both be greater than or equal to 0 and less than or equal to the height of the specified srcSubresource of srcImage

- VUID-VkBlitImageInfo2KHR-srcImage-00245
  If srcImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, srcOffsets[0].y must be 0 and srcOffsets[1].y must be 1

- VUID-VkBlitImageInfo2KHR-srcOffset-00246
  For each element of pRegions, srcOffsets[0].z and srcOffsets[1].z must both be greater than or equal to 0 and less than or equal to the depth of the specified srcSubresource of srcImage

- VUID-VkBlitImageInfo2KHR-srcImage-00247
  If srcImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, srcOffsets[0].z must be 0 and srcOffsets[1].z must be 1

- VUID-VkBlitImageInfo2KHR-dstOffset-00248
  For each element of pRegions, dstOffsets[0].x and dstOffsets[1].x must both be greater than or equal to 0 and less than or equal to the width of the specified dstSubresource of dstImage

- VUID-VkBlitImageInfo2KHR-dstOffset-00249
  For each element of pRegions, dstOffsets[0].y and dstOffsets[1].y must both be greater than or equal to 0 and less than or equal to the height of the specified dstSubresource of dstImage

- VUID-VkBlitImageInfo2KHR-dstImage-00250
  If dstImage is of type VK_IMAGE_TYPE_1D, then for each element of pRegions, dstOffsets[0].y must be 0 and dstOffsets[1].y must be 1

- VUID-VkBlitImageInfo2KHR-dstOffset-00251
  For each element of pRegions, dstOffsets[0].z and dstOffsets[1].z must both be greater than or equal to 0 and less than or equal to the depth of the specified dstSubresource of dstImage

- VUID-VkBlitImageInfo2KHR-dstImage-00252
  If dstImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then for each element of pRegions, dstOffsets[0].z must be 0 and dstOffsets[1].z must be 1

- VUID-VkBlitImageInfo2KHR-pRegions-04561
If any element of `pRegions` contains `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, then `srcImage` and `dstImage` must not be a block-compressed image.

- VUID-VkBlitImageInfo2KHR-pRegions-04562

If any element of `pRegions` contains `VkCopyCommandTransformInfoQCOM` in its `pNext` chain, then the `srcImage` must be of type `VK_IMAGE_TYPE_2D` and must not be a multi-planar format.

**Valid Usage (Implicit)**

- VUID-VkBlitImageInfo2KHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2_KHR`

- VUID-VkBlitImageInfo2KHR-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkBlitImageInfo2KHR-srcImage-parameter
  `srcImage` must be a valid `VkImage` handle

- VUID-VkBlitImageInfo2KHR-srcImageLayout-parameter
  `srcImageLayout` must be a valid `VkImageLayout` value

- VUID-VkBlitImageInfo2KHR-dstImage-parameter
  `dstImage` must be a valid `VkImage` handle

- VUID-VkBlitImageInfo2KHR-dstImageLayout-parameter
  `dstImageLayout` must be a valid `VkImageLayout` value

- VUID-VkBlitImageInfo2KHR-pRegions-parameter
  `pRegions` must be a valid pointer to an array of `regionCount` valid `VkImageBlit2KHR` structures

- VUID-VkBlitImageInfo2KHR-filter-parameter
  `filter` must be a valid `VkFilter` value

- VUID-VkBlitImageInfo2KHR-regionCount-arraylength
  `regionCount` must be greater than 0

Both of `dstImage`, and `srcImage` must have been created, allocated, or retrieved from the same `VkDevice`

The `VkImageBlit2KHR` structure is defined as:
```c
typedef struct VkImageBlit2KHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkImageSubresourceLayers    srcSubresource;
    VkOffset3D                  srcOffsets[2];
    VkImageSubresourceLayers    dstSubresource;
    VkOffset3D                  dstOffsets[2];
} VkImageBlit2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcSubresource` is the subresource to blit from.
- `srcOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the source region within `srcSubresource`.
- `dstSubresource` is the subresource to blit into.
- `dstOffsets` is a pointer to an array of two `VkOffset3D` structures specifying the bounds of the destination region within `dstSubresource`.

For each element of the `pRegions` array, a blit operation is performed for the specified source and destination regions.

---

**Valid Usage**

- **VUID-VkImageBlit2KHR-aspectMask-00238**
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must match
- **VUID-VkImageBlit2KHR-layerCount-00239**
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match
Valid Usage (Implicit)

- **VUID-VkImageBlit2KHR-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_IMAGE_BLIT_2_KHR`

- **VUID-VkImageBlit2KHR-pNext-pNext**
  - `pNext` must be `NULL` or a pointer to a valid instance of `VkCopyCommandTransformInfoQCOM`

- **VUID-VkImageBlit2KHR-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkImageBlit2KHR-srcSubresource-parameter**
  - `srcSubresource` must be a valid `VkImageSubresourceLayers` structure

- **VUID-VkImageBlit2KHR-dstSubresource-parameter**
  - `dstSubresource` must be a valid `VkImageSubresourceLayers` structure

For `vkCmdBlitImage2KHR`, each region copied can include a rotation. To specify a rotated region, add `VkCopyCommandTransformInfoQCOM` to the `pNext` chain of `VkImageBlit2KHR`. For each region with a rotation specified, Image Blits with Scaling and Rotation specifies how coordinates are rotated prior to sampling from the source image. When rotation is specified, the source and destination images must be a 2D image and must not be blocked image or multi-planar format.

<<formats-requiring-sampler-ycbcr-conversion,_multi-planar_image>>, or specify a destination <<blocked-image, blocked image>>.

### 20.5.1. Image Blits with Scaling and Rotation

When `VkCopyCommandTransformInfoQCOM` is in the `pNext` chain of `VkImageBlit2KHR`, the specified region is rotated during the blit. The following description of rotated addressing replaces the description in `vkCmdBlitImage`.

The following code computes rotation of normalized coordinates.
/ rotation of normalized coordinates
VkOffset2D RotateNormUV(VkOffset2D in, VkSurfaceTransformFlagBitsKHR flags)
{
    VkOffset2D output;
    switch (flags)
    {
        case VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR:
            out.x = in.x;
            out.y = in.y;
            break;
        case VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR:
            out.x = in.y;
            out.y = 1.0 - in.x;
            break;
        case VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR:
            out.x = 1.0 - in.x;
            out.y = 1.0 - in.y;
            break;
        case VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR:
            out.x = 1.0 - in.y;
            out.y = in.x;
            break;
    }
    return out;
}

- For each destination texel, the integer coordinate of that texel is converted to an unnormalized texture coordinate, using the effective inverse of the equations described in unnormalized to integer conversion:

\[
    u_{\text{base}} = i + \frac{1}{2} \\
    v_{\text{base}} = j + \frac{1}{2} \\
    w_{\text{base}} = k + \frac{1}{2}
\]

- These base coordinates are then offset by the first destination offset:

\[
    u_{\text{offset}} = u_{\text{base}} - x_{\text{dst0}} \\
    v_{\text{offset}} = v_{\text{base}} - y_{\text{dst0}} \\
    w_{\text{offset}} = w_{\text{base}} - z_{\text{dst0}}
\]
\[ a_{\text{offset}} = a - \text{baseArrayCount}_{\text{dst}} \]

- The UV destination coordinates are scaled by the destination region, rotated, and scaled by the source region.

\[
\begin{align*}
  u_{\text{dest} \text{ scaled}} &= u_{\text{offset}} / (x_{\text{dst}1} - x_{\text{dst}0}) \\
  v_{\text{dest} \text{ scaled}} &= v_{\text{offset}} / (y_{\text{dst}1} - y_{\text{dst}0})
\end{align*}
\]

\[
(u_{\text{src} \text{ scaled}}, v_{\text{src} \text{ scaled}}) = \text{RotateNormUV}(u_{\text{dest} \text{ scaled}}, v_{\text{dest} \text{ scaled}}, \text{transform})
\]

\[
\begin{align*}
  u_{\text{scaled}} &= u_{\text{src} \text{ scaled}} \times (x_{\text{Src}1} - x_{\text{Src}0}) \\
  v_{\text{scaled}} &= v_{\text{src} \text{ scaled}} \times (y_{\text{Src}1} - y_{\text{Src}0})
\end{align*}
\]

- The W coordinate is unaffected by rotation. The scale is determined from the ratio of source and destination regions, and applied to the offset coordinate:

\[
\begin{align*}
  \text{scale}_w &= (z_{\text{Src}1} - z_{\text{Src}0}) / (z_{\text{dst}1} - z_{\text{dst}0}) \\
  w_{\text{scaled}} &= w_{\text{offset}} \times \text{scale}_w
\end{align*}
\]

- Finally the source offset is added to the scaled source coordinates, to determine the final unnormalized coordinates used to sample from srcImage:

\[
\begin{align*}
  u &= u_{\text{scaled}} + x_{\text{Src}0} \\
  v &= v_{\text{scaled}} + y_{\text{Src}0} \\
  w &= w_{\text{scaled}} + z_{\text{Src}0} \\
  q &= \text{mipLevel}
\end{align*}
\]

\[ a = a_{\text{offset}} + \text{baseArrayCount}_{\text{src}} \]

These coordinates are used to sample from the source image, as described in Image Operations.
chapter, with the filter mode equal to that of `filter`, a mipmap mode of `VK_SAMPLER_MIPMAP_MODE_NEAREST` and an address mode of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`. Implementations must clamp at the edge of the source image, and may additionally clamp to the edge of the source region.

### 20.6. Resolving Multisample Images

To resolve a multisample color image to a non-multisample color image, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdResolveImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkImageResolve*                       pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the resolve.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the resolve.
- `regionCount` is the number of regions to resolve.
- `pRegions` is a pointer to an array of `VkImageResolve` structures specifying the regions to resolve.

During the resolve the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination. If the source formats are floating-point or normalized types, the sample values for each pixel are resolved in an implementation-dependent manner. If the source formats are integer types, a single sample’s value is selected for each pixel.

`srcOffset` and `dstOffset` select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data. `extent` is the size in texels of the source image to resolve in width, height and depth. Each element of `pRegions` must be a region that is contained within its corresponding image.

Resolves are done layer by layer starting with `baseArrayLayer` member of `srcSubresource` for the source and `dstSubresource` for the destination. `layerCount` layers are resolved to the destination image.
Valid Usage

• VUID-vkCmdResolveImage-commandBuffer-01837
  If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image

• VUID-vkCmdResolveImage-commandBuffer-01838
  If `commandBuffer` is an unprotected command buffer, then `dstImage` must not be a protected image

• VUID-vkCmdResolveImage-commandBuffer-01839
  If `commandBuffer` is a protected command buffer, then `dstImage` must not be an unprotected image

• VUID-vkCmdResolveImage-pRegions-00255
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory

• VUID-vkCmdResolveImage-srcImage-00256
  If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

• VUID-vkCmdResolveImage-srcImage-00257
  `srcImage` must have a sample count equal to any valid sample count value other than `VK_SAMPLE_COUNT_1_BIT`

• VUID-vkCmdResolveImage-dstImage-00258
  If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

• VUID-vkCmdResolveImage-dstImage-00259
  `dstImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

• VUID-vkCmdResolveImage-srcImageLayout-00260
  `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

• VUID-vkCmdResolveImage-srcImageLayout-01400
  `srcImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

• VUID-vkCmdResolveImage-dstImageLayout-00262
  `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

• VUID-vkCmdResolveImage-dstImageLayout-01401
  `dstImageLayout` must be `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`, `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`

• VUID-vkCmdResolveImage-dstImage-02003
  The format features of `dstImage` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`

• VUID-vkCmdResolveImage-srcImage-01386
  `srcImage` and `dstImage` must have been created with the same image format

• VUID-vkCmdResolveImage-srcSubresource-01709
The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.

- VUID-vkCmdResolveImage-dstSubresource-01710
  The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.

- VUID-vkCmdResolveImage-srcSubresource-01711
  The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created.

- VUID-vkCmdResolveImage-dstSubresource-01712
  The `dstSubresource.baseArrayLayer + dstSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `dstImage` was created.

- VUID-vkCmdResolveImage-dstImage-02546
  `dstImage` and `srcImage` **must** not have been created with flags containing `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`.

- VUID-vkCmdResolveImage-srcImage-04446
  If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `srcSubresource.baseArrayLayer` **must** be 0 and `srcSubresource.layerCount` **must** be 1.

- VUID-vkCmdResolveImage-srcImage-04447
  If either `srcImage` or `dstImage` are of type `VK_IMAGE_TYPE_3D`, then for each element of `pRegions`, `dstSubresource.baseArrayLayer` **must** be 0 and `dstSubresource.layerCount` **must** be 1.

- VUID-vkCmdResolveImage-srcOffset-00269
  For each element of `pRegions`, `srcOffset.x` and `(extent.width + srcOffset.x)` **must** both be greater than or equal to 0 and less than or equal to the width of the specified `srcSubresource` of `srcImage`.

- VUID-vkCmdResolveImage-srcOffset-00270
  For each element of `pRegions`, `srcOffset.y` and `(extent.height + srcOffset.y)` **must** both be greater than or equal to 0 and less than or equal to the height of the specified `srcSubresource` of `srcImage`.

- VUID-vkCmdResolveImage-srcOffset-00271
  If `srcImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `srcOffset.y` **must** be 0 and `extent.height` **must** be 1.

- VUID-vkCmdResolveImage-srcOffset-00272
  For each element of `pRegions`, `srcOffset.z` and `(extent.depth + srcOffset.z)` **must** both be greater than or equal to 0 and less than or equal to the depth of the specified `srcSubresource` of `srcImage`.

- VUID-vkCmdResolveImage-srcOffset-00273
  If `srcImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `srcOffset.z` **must** be 0 and `extent.depth` **must** be 1.

- VUID-vkCmdResolveImage-dstOffset-00274
For each element of `pRegions`, `dstOffset.x` and `(extent.width + dstOffset.x)` must both be greater than or equal to 0 and less than or equal to the width of the specified `dstSubresource` of `dstImage`.

- VUID-vkCmdResolveImage-dstOffset-00275
  For each element of `pRegions`, `dstOffset.y` and `(extent.height + dstOffset.y)` must both be greater than or equal to 0 and less than or equal to the height of the specified `dstSubresource` of `dstImage`.

- VUID-vkCmdResolveImage-dstImage-00276
  If `dstImage` is of type `VK_IMAGE_TYPE_1D`, then for each element of `pRegions`, `dstOffset.y` must be 0 and `extent.height` must be 1.

- VUID-vkCmdResolveImage-dstOffset-00277
  For each element of `pRegions`, `dstOffset.z` and `(extent.depth + dstOffset.z)` must both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`.

- VUID-vkCmdResolveImage-dstImage-00278
  If `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffset.z` must be 0 and `extent.depth` must be 1.
Valid Usage (Implicit)

- **VUID-vkCmdResolveImage-commandBuffer-parameter**
  \(\text{commandBuffer}\) must be a valid \text{VkCommandBuffer} handle

- **VUID-vkCmdResolveImage-srcImage-parameter**
  \(\text{srcImage}\) must be a valid \text{VkImage} handle

- **VUID-vkCmdResolveImage-srcImageLayout-parameter**
  \(\text{srcImageLayout}\) must be a valid \text{VkImageLayout} value

- **VUID-vkCmdResolveImage-dstImage-parameter**
  \(\text{dstImage}\) must be a valid \text{VkImage} handle

- **VUID-vkCmdResolveImage-dstImageLayout-parameter**
  \(\text{dstImageLayout}\) must be a valid \text{VkImageLayout} value

- **VUID-vkCmdResolveImage-pRegions-parameter**
  \(\text{pRegions}\) must be a valid pointer to an array of \text{regionCount} valid \text{VkImageResolve} structures

- **VUID-vkCmdResolveImage-commandBuffer-recording**
  \(\text{commandBuffer}\) must be in the \text{recording state}

- **VUID-vkCmdResolveImage-commandBuffer-cmdpool**
  The \text{VkCommandPool} that \(\text{commandBuffer}\) was allocated from must support graphics operations

- **VUID-vkCmdResolveImage-renderpass**
  This command must only be called outside of a render pass instance

- **VUID-vkCmdResolveImage-regionCount-arraylength**
  \(\text{regionCount}\) must be greater than 0

- **VUID-vkCmdResolveImage-commonparent**
  Each of \(\text{commandBuffer}\), \(\text{dstImage}\), and \(\text{srcImage}\) must have been created, allocated, or retrieved from the same \text{VkDevice}

Host Synchronization

- Host access to \(\text{commandBuffer}\) must be externally synchronized

- Host access to the \text{VkCommandPool} that \(\text{commandBuffer}\) was allocated from must be externally synchronized
The `VkImageResolve` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageResolve {
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageResolve;
```

- `srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.
- `srcOffset` and `dstOffset` select the initial `x`, `y`, and `z` offsets in texels of the sub-regions of the source and destination image data.
- `extent` is the size in texels of the source image to resolve in `width`, `height` and `depth`.

### Valid Usage

- **VUID-VkImageResolve-aspectMask-00266**
  The `aspectMask` member of `srcSubresource` and `dstSubresource` must only contain `VK_IMAGE_ASPECT_COLOR_BIT`

- **VUID-VkImageResolve-layerCount-00267**
  The `layerCount` member of `srcSubresource` and `dstSubresource` must match

### Valid Usage (Implicit)

- **VUID-VkImageResolve-srcSubresource-parameter**
  `srcSubresource` must be a valid `VkImageSubresourceLayers` structure

- **VUID-VkImageResolve-dstSubresource-parameter**
  `dstSubresource` must be a valid `VkImageSubresourceLayers` structure
20.7. Buffer Markers

To write a 32-bit marker value into a buffer as a pipelined operation, call:

```c
// Provided by VK_KHR_synchronization2 with VK_AMD_buffer_marker
void vkCmdWriteBufferMarker2AMD(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlags2KHR stage,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    uint32_t marker);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `stage` specifies the pipeline stage whose completion triggers the marker write.
- `dstBuffer` is the buffer where the marker will be written.
- `dstOffset` is the byte offset into the buffer where the marker will be written.
- `marker` is the 32-bit value of the marker.

The command will write the 32-bit marker value into the buffer only after all preceding commands have finished executing up to at least the specified pipeline stage. This includes the completion of other preceding `vkCmdWriteBufferMarker2AMD` commands so long as their specified pipeline stages occur either at the same time or earlier than this command's specified `pipelineStage`.

While consecutive buffer marker writes with the same `pipelineStage` parameter implicitly complete in submission order, memory and execution dependencies between buffer marker writes and other operations must still be explicitly ordered using synchronization commands. The access scope for buffer marker writes falls under the `VK_ACCESS_TRANSFER_WRITE_BIT`, and the pipeline stages for identifying the synchronization scope must include both `pipelineStage` and `VK_PIPELINE_STAGE_TRANSFER_BIT`.

**Note**

Similar to `vkCmdWriteTimestamp2KHR`, if an implementation is unable to write a marker at any specific pipeline stage, it may instead do so at any logically later stage.

**Note**

Implementations may only support a limited number of pipelined marker write operations in flight at a given time. Thus an excessive number of marker write operations may degrade command execution performance.
Valid Usage

- If the geometry shaders feature is not enabled, **stage** must not contain `VK_PIPELINE_STAGE_2_GEOMETRY_SHADER_BIT_KHR`.
- If the tessellation shaders feature is not enabled, **stage** must not contain `VK_PIPELINE_STAGE_2_TESSELLATION_CONTROL_SHADER_BIT_KHR` or `VK_PIPELINE_STAGE_2_TESSELLATION_EVALUATION_SHADER_BIT_KHR`.
- If the conditional rendering feature is not enabled, **stage** must not contain `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`.
- If the fragment density map feature is not enabled, **stage** must not contain `VK_PIPELINE_STAGE_2.Fragment_density_process_BIT_EXT`.
- If the task shaders feature is not enabled, **stage** must not contain `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`.
- If the shading rate image feature is not enabled, **stage** must not contain `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`.

The synchronization2 feature must be enabled.

**stage** must include only a single pipeline stage.

**stage** must include only stages that are valid for the queue family that was used to create the command pool that `commandBuffer` was allocated from.

**dstOffset** must be less than or equal to the size of `dstBuffer` minus 4.

**dstBuffer** must have been created with the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

If `dstBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

**dstOffset** must be a multiple of 4.
Valid Usage (Implicit)

- VUID-vkCmdWriteBufferMarker2AMD-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdWriteBufferMarker2AMD-stage-parameter
  stage must be a valid combination of VkPipelineStageFlagBits2KHR values

- VUID-vkCmdWriteBufferMarker2AMD-stage-requiredmask
  stage must not be 0

- VUID-vkCmdWriteBufferMarker2AMD-dstBuffer-parameter
  dstBuffer must be a valid VkBuffer handle

- VUID-vkCmdWriteBufferMarker2AMD-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdWriteBufferMarker2AMD-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations

- VUID-vkCmdWriteBufferMarker2AMD-commonparent
  Both of commandBuffer, and dstBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Transfer Graphics</td>
<td>Transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To write a 32-bit marker value into a buffer as a pipelined operation, call:
// Provided by VK_AMD_buffer_marker
void vkCmdWriteBufferMarkerAMD(
    VkCommandBuffer commandBuffer,
    VkPipelineStageFlagBits pipelineStage,
    VkBuffer dstBuffer,
    VkDeviceSize dstOffset,
    uint32_t marker);

- **commandBuffer** is the command buffer into which the command will be recorded.
- **pipelineStage** is one of the VkPipelineStageFlagBits values, specifying the pipeline stage whose completion triggers the marker write.
- **dstBuffer** is the buffer where the marker will be written to.
- **dstOffset** is the byte offset into the buffer where the marker will be written to.
- **marker** is the 32-bit value of the marker.

The command will write the 32-bit marker value into the buffer only after all preceding commands have finished executing up to at least the specified pipeline stage. This includes the completion of other preceding vkCmdWriteBufferMarkerAMD commands so long as their specified pipeline stages occur either at the same time or earlier than this command's specified pipelineStage.

While consecutive buffer marker writes with the same pipelineStage parameter are implicitly complete in submission order, memory and execution dependencies between buffer marker writes and other operations must still be explicitly ordered using synchronization commands. The access scope for buffer marker writes falls under the VK_ACCESS_TRANSFER_WRITE_BIT, and the pipeline stages for identifying the synchronization scope must include both pipelineStage and VK_PIPELINE_STAGE_TRANSFER_BIT.

**Note**

Similar to vkCmdWriteTimestamp, if an implementation is unable to write a marker at any specific pipeline stage, it may instead do so at any logically later stage.

**Note**

Implementations may only support a limited number of pipelined marker write operations in flight at a given time, thus excessive number of marker write operations may degrade command execution performance.
Valid Usage

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04074
  pipelineStage must be a valid stage for the queue family that was used to create the command pool that commandBuffer was allocated from

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04075
  If the geometry shaders feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04076
  If the tessellation shaders feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT or VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04077
  If the conditional rendering feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04078
  If the fragment density map feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04079
  If the transform feedback feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04080
  If the mesh shaders feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV or VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV

- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-04081
  If the shading rate image feature is not enabled, pipelineStage must not be VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV

- VUID-vkCmdWriteBufferMarkerAMD-dstOffset-01798
  dstOffset must be less than or equal to the size of dstBuffer minus 4

- VUID-vkCmdWriteBufferMarkerAMD-dstBuffer-01799
  dstBuffer must have been created with VK_BUFFER_USAGE_TRANSFER_DST_BIT usage flag

- VUID-vkCmdWriteBufferMarkerAMD-dstBuffer-01800
  If dstBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdWriteBufferMarkerAMD-dstOffset-01801
  dstOffset must be a multiple of 4
Valid Usage (Implicit)

- VUID-vkCmdWriteBufferMarkerAMD-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdWriteBufferMarkerAMD-pipelineStage-parameter
  - `pipelineStage` must be a valid `VkPipelineStageFlagBits` value
- VUID-vkCmdWriteBufferMarkerAMD-dstBuffer-parameter
  - `dstBuffer` must be a valid `VkBuffer` handle
- VUID-vkCmdWriteBufferMarkerAMD-commandBuffer-recording
  - `commandBuffer` must be in the recording state
- VUID-vkCmdWriteBufferMarkerAMD-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- VUID-vkCmdWriteBufferMarkerAMD-commonparent
  - Both of `commandBuffer`, and `dstBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Transfer, Graphics, Compute</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A more extensible version of the resolve image command is defined below.

To resolve a multisample image to a non-multisample image, call:

```c
// Provided by VK_KHR_copy_commands2
void vkCmdResolveImage2KHR(
    VkCommandBuffer                             commandBuffer,
    const VkResolveImageInfo2KHR*               pResolveImageInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- **pResolveImageInfo** is a pointer to a `VkResolveImageInfo2KHR` structure describing the resolve parameters.

This command is functionally identical to `vkCmdResolveImage`, but includes extensible substructures that include `sType` and `pNext` parameters, allowing them to be more easily extended.

**Valid Usage**

- **VUID-vkCmdResolveImage2KHR-commandBuffer-01837**  
  If `commandBuffer` is an unprotected command buffer, then `srcImage` must not be a protected image

- **VUID-vkCmdResolveImage2KHR-commandBuffer-01838**  
  If `commandBuffer` is an unprotected command buffer, then `dstImage` must not be a protected image

- **VUID-vkCmdResolveImage2KHR-commandBuffer-01839**  
  If `commandBuffer` is a protected command buffer, then `dstImage` must not be an unprotected image

**Valid Usage (Implicit)**

- **VUID-vkCmdResolveImage2KHR-commandBuffer-parameter**  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdResolveImage2KHR-pResolveImageInfo-parameter**  
  `pResolveImageInfo` must be a valid pointer to a valid `VkResolveImageInfo2KHR` structure

- **VUID-vkCmdResolveImage2KHR-commandBuffer-recording**  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdResolveImage2KHR-commandBuffer-cmdpool**  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- **VUID-vkCmdResolveImage2KHR-renderpass**  
  This command must only be called outside of a render pass instance

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
The `VkResolveImageInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_copy_commands2
typedef struct VkResolveImageInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImage srcImage;
    VkImageLayout srcImageLayout;
    VkImage dstImage;
    VkImageLayout dstImageLayout;
    uint32_t regionCount;
    const VkImageResolve2KHR* pRegions;
} VkResolveImageInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcImage` is the source image.
- `srcImageLayout` is the layout of the source image subresources for the resolve.
- `dstImage` is the destination image.
- `dstImageLayout` is the layout of the destination image subresources for the resolve.
- `regionCount` is the number of regions to resolve.
- `pRegions` is a pointer to an array of `VkImageResolve2KHR` structures specifying the regions to resolve.
Valid Usage

- VUID-VkResolveImageInfo2KHR-pRegions-00255
  The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, **must** not overlap in memory

- VUID-VkResolveImageInfo2KHR-srcImage-00256
  If `srcImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkResolveImageInfo2KHR-srcImage-00257
  `srcImage` **must** have a sample count equal to any valid sample count value other than `VK_SAMPLE_COUNT_1_BIT`

- VUID-VkResolveImageInfo2KHR-dstImage-00258
  If `dstImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-VkResolveImageInfo2KHR-dstImage-00259
  `dstImage` **must** have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`

- VUID-VkResolveImageInfo2KHR-srcImageLayout-01400
  `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-VkResolveImageInfo2KHR-dstImageLayout-00262
  `dstImageLayout` **must** specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`

- VUID-VkResolveImageInfo2KHR-dstImage-02003
  The format features of `dstImage` **must** contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`

- VUID-VkResolveImageInfo2KHR-srcImage-01386
  `srcImage` and `dstImage` **must** have been created with the same image format

- VUID-VkResolveImageInfo2KHR-srcSubresource-01709
  The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created

- VUID-VkResolveImageInfo2KHR-dstSubresource-01710
  The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created

- VUID-VkResolveImageInfo2KHR-srcSubresource-01711
  The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo` when `srcImage` was created
The \( \text{dstSubresource.baseArrayLayer} + \text{dstSubresource.layerCount} \) of each element of \( \text{pRegions} \) must be less than or equal to the \( \text{arrayLayers} \) specified in \( \text{VkImageCreateInfo} \) when \( \text{dstImage} \) was created.

\( \text{dstImage} \) and \( \text{srcImage} \) must not have been created with flags containing \( \text{VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT} \).

If either \( \text{srcImage} \) or \( \text{dstImage} \) are of type \( \text{VK_IMAGE_TYPE_3D} \), then for each element of \( \text{pRegions} \), \( \text{srcSubresource.baseArrayLayer} \) must be 0 and \( \text{srcSubresource.layerCount} \) must be 1.

If either \( \text{srcImage} \) or \( \text{dstImage} \) are of type \( \text{VK_IMAGE_TYPE_3D} \), then for each element of \( \text{pRegions} \), \( \text{dstSubresource.baseArrayLayer} \) must be 0 and \( \text{dstSubresource.layerCount} \) must be 1.

For each element of \( \text{pRegions} \), \( \text{srcOffset.x} \) and \( (\text{extent.width} + \text{srcOffset.x}) \) must both be greater than or equal to 0 and less than or equal to the width of the specified \( \text{srcSubresource} \) of \( \text{srcImage} \).

For each element of \( \text{pRegions} \), \( \text{srcOffset.y} \) and \( (\text{extent.height} + \text{srcOffset.y}) \) must both be greater than or equal to 0 and less than or equal to the height of the specified \( \text{srcSubresource} \) of \( \text{srcImage} \).

If \( \text{srcImage} \) is of type \( \text{VK_IMAGE_TYPE_1D} \), then for each element of \( \text{pRegions} \), \( \text{srcOffset.y} \) must be 0 and \( \text{extent.height} \) must be 1.

For each element of \( \text{pRegions} \), \( \text{srcOffset.z} \) and \( (\text{extent.depth} + \text{srcOffset.z}) \) must both be greater than or equal to 0 and less than or equal to the depth of the specified \( \text{srcSubresource} \) of \( \text{srcImage} \).

If \( \text{srcImage} \) is of type \( \text{VK_IMAGE_TYPE_1D} \) or \( \text{VK_IMAGE_TYPE_2D} \), then for each element of \( \text{pRegions} \), \( \text{srcOffset.z} \) must be 0 and \( \text{extent.depth} \) must be 1.

For each element of \( \text{pRegions} \), \( \text{dstOffset.x} \) and \( (\text{extent.width} + \text{dstOffset.x}) \) must both be greater than or equal to 0 and less than or equal to the width of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \).

For each element of \( \text{pRegions} \), \( \text{dstOffset.y} \) and \( (\text{extent.height} + \text{dstOffset.y}) \) must both be greater than or equal to 0 and less than or equal to the height of the specified \( \text{dstSubresource} \) of \( \text{dstImage} \).

If \( \text{dstImage} \) is of type \( \text{VK_IMAGE_TYPE_1D} \), then for each element of \( \text{pRegions} \), \( \text{dstOffset.y} \) must be 0 and \( \text{extent.height} \) must be 1.
For each element of `pRegions`, `dstOffset.z` and (`extent.depth` + `dstOffset.z`) **must** both be greater than or equal to 0 and less than or equal to the depth of the specified `dstSubresource` of `dstImage`.

If `dstImage` is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then for each element of `pRegions`, `dstOffset.z` **must** be 0 and `extent.depth` **must** be 1.

**Valid Usage (Implicit)**

- **VUID-VkResolveImageInfo2KHR-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2_KHR`

- **VUID-VkResolveImageInfo2KHR-pNext-pNext**
  
  `pNext` **must** be `NULL`

- **VUID-VkResolveImageInfo2KHR-srcImage-parameter**
  
  `srcImage` **must** be a valid `VkImage` handle

- **VUID-VkResolveImageInfo2KHR-srcImageLayout-parameter**
  
  `srcImageLayout` **must** be a valid `VkImageLayout` value

- **VUID-VkResolveImageInfo2KHR-dstImage-parameter**
  
  `dstImage` **must** be a valid `VkImage` handle

- **VUID-VkResolveImageInfo2KHR-dstImageLayout-parameter**
  
  `dstImageLayout` **must** be a valid `VkImageLayout` value

- **VUID-VkResolveImageInfo2KHR-pRegions-parameter**
  
  `pRegions` **must** be a valid pointer to an array of `regionCount` valid `VkImageResolve2KHR` structures

- **VUID-VkResolveImageInfo2KHR-regionCount-arraylength**
  
  `regionCount` **must** be greater than 0

Both of `dstImage`, and `srcImage` **must** have been created, allocated, or retrieved from the same `VkDevice`.

The `VkImageResolve2KHR` structure is defined as:

---

1290 | Chapter 20. Copy Commands
typedef struct VkImageResolve2KHR {
    VkStructureType sType;
    const void* pNext;
    VkImageSubresourceLayers srcSubresource;
    VkOffset3D srcOffset;
    VkImageSubresourceLayers dstSubresource;
    VkOffset3D dstOffset;
    VkExtent3D extent;
} VkImageResolve2KHR;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **srcSubresource** and **dstSubresource** are *VkImageSubresourceLayers* structures specifying the image subresources of the images used for the source and destination image data, respectively. Resolve of depth/stencil images is not supported.
• **srcOffset** and **dstOffset** select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
• **extent** is the size in texels of the source image to resolve in width, height and depth.

### Valid Usage

- **VUID-VkImageResolve2KHR-aspectMask-00266**
  
  The *aspectMask* member of **srcSubresource** and **dstSubresource** must only contain VK_IMAGE_ASPECT_COLOR_BIT

- **VUID-VkImageResolve2KHR-layerCount-00267**
  
  The *layerCount* member of **srcSubresource** and **dstSubresource** must match

### Valid Usage (Implicit)

- **VUID-VkImageResolve2KHR-sType-sType**
  
  **sType** must be VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2_KHR

- **VUID-VkImageResolve2KHR-pNext-pNext**
  
  **pNext** must be NULL

- **VUID-VkImageResolve2KHR-srcSubresource-parameter**
  
  **srcSubresource** must be a valid *VkImageSubresourceLayers* structure

- **VUID-VkImageResolve2KHR-dstSubresource-parameter**
  
  **dstSubresource** must be a valid *VkImageSubresourceLayers* structure
Chapter 21. Drawing Commands

Drawing commands (commands with Draw in the name) provoke work in a graphics pipeline. Drawing commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound graphics pipeline. A graphics pipeline must be bound to a command buffer before any drawing commands are recorded in that command buffer.

Drawing can be achieved in two modes:

- **Programmable Mesh Shading**, the mesh shader assembles primitives, or
- **Programmable Primitive Shading**, the input primitives are assembled

as follows.

Each draw is made up of zero or more vertices and zero or more instances, which are processed by the device and result in the assembly of primitives. Primitives are assembled according to the pInputAssemblyState member of the VkGraphicsPipelineCreateInfo structure, which is of type VkPipelineInputAssemblyStateCreateInfo:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineInputAssemblyStateCreateFlags    flags;
    VkPrimitiveTopology                        topology;
    VkBool32                                   primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **topology** is a VkPrimitiveTopology defining the primitive topology, as described below.
- **primitiveRestartEnable** controls whether a special vertex index value is treated as restarting the assembly of primitives. This enable only applies to indexed draws (vkCmdDrawIndexed and vkCmdDrawIndexedIndirect), and the special index value is either 0xFFFFFFFF when the indexType parameter of vkCmdBindIndexBuffer is equal to VK_INDEX_TYPE_UINT32, 0xFF when indexType is equal to VK_INDEX_TYPE_UINT8_EXT, or 0xFFFF when indexType is equal to VK_INDEX_TYPE_UINT16. Primitive restart is not allowed for “list” topologies.

Restarting the assembly of primitives discards the most recent index values if those elements formed an incomplete primitive, and restarts the primitive assembly using the subsequent indices, but only assembling the immediately following element through the end of the originally specified elements. The primitive restart index value comparison is performed before adding the vertexOffset value to the index value.
Valid Usage

- **VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00428**
  If `topology` is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, `primitiveRestartEnable` must be `VK_FALSE`.

- **VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00429**
  If the geometry shaders feature is not enabled, `topology` must not be any of `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`.

- **VUID-VkPipelineInputAssemblyStateCreateInfo-topology-00430**
  If the tessellation shaders feature is not enabled, `topology` must not be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`.

- **VUID-VkPipelineInputAssemblyStateCreateInfo-triangleFans-04452**
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::triangleFans` is `VK_FALSE`, `topology` must not be `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`.

Valid Usage (Implicit)

- **VUID-VkPipelineInputAssemblyStateCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO`.

- **VUID-VkPipelineInputAssemblyStateCreateInfo-pNext-pNext**
  `pNext` must be `NULL`.

- **VUID-VkPipelineInputAssemblyStateCreateInfo-flags-zerobitmask**
  `flags` must be `0`.

- **VUID-VkPipelineInputAssemblyStateCreateInfo-topology-parameter**
  `topology` must be a valid `VkPrimitiveTopology` value.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineInputAssemblyStateCreateFlags;
```

`VkPipelineInputAssemblyStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

To dynamically control whether a special vertex index value is treated as restarting the assembly of primitives:
void vkCmdSetPrimitiveRestartEnableEXT(
    VkCommandBuffer commandBuffer,
    VkBool32 primitiveRestartEnable);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `primitiveRestartEnable` controls whether a special vertex index value is treated as restarting the assembly of primitives. It behaves in the same way as `VkPipelineInputAssemblyStateCreateInfo::primitiveRestartEnable`.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage

- VUID-vkCmdSetPrimitiveRestartEnableEXT-None-04866
  The `extendedDynamicState2` feature **must** be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetPrimitiveRestartEnableEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdSetPrimitiveRestartEnableEXT-commandBuffer-recording
  `commandBuffer` **must** be in the `recording` state
- VUID-vkCmdSetPrimitiveRestartEnableEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized
21.1. Primitive Topologies

*Primitive topology* determines how consecutive vertices are organized into primitives, and determines the type of primitive that is used at the beginning of the graphics pipeline. The effective topology for later stages of the pipeline is altered by tessellation or geometry shading (if either is in use) and depends on the execution modes of those shaders. In the case of mesh shading the only effective topology is defined by the execution mode of the mesh shader.

The primitive topologies defined by `VkPrimitiveTopology` are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;
```

- `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` specifies a series of separate point primitives.
- `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` specifies a series of separate line primitives.
- `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP` specifies a series of connected line primitives with consecutive lines sharing a vertex.
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST` specifies a series of separate triangle primitives.
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP` specifies a series of connected triangle primitives with consecutive triangles sharing an edge.
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN` specifies a series of connected triangle primitives with all triangles sharing a common vertex. If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::triangleFans` is `VK_FALSE`, then triangle fans are not supported by the implementation, and `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN` must not be used.
• **VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY** specifies a series of separate line primitives with adjacency.

• **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY** specifies a series of connected line primitives with adjacency, with consecutive primitives sharing three vertices.

• **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY** specifies a series of separate triangle primitives with adjacency.

• **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY** specifies connected triangle primitives with adjacency, with consecutive triangles sharing an edge.

• **VK_PRIMITIVE_TOPOLOGY_PATCH_LIST** specifies separate patch primitives.

Each primitive topology, and its construction from a list of vertices, is described in detail below with a supporting diagram, according to the following key:

<table>
<thead>
<tr>
<th></th>
<th>Vertex</th>
<th>A point in 3-dimensional space. Positions chosen within the diagrams are arbitrary and for illustration only.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Vertex Number</td>
<td>Sequence position of a vertex within the provided vertex data.</td>
</tr>
<tr>
<td></td>
<td>Provoking Vertex</td>
<td>Provoking vertex within the main primitive. The tail is angled towards the relevant primitive. Used in flat shading.</td>
</tr>
<tr>
<td></td>
<td>Primitive Edge</td>
<td>An edge connecting the points of a main primitive.</td>
</tr>
<tr>
<td></td>
<td>Adjacency Edge</td>
<td>Points connected by these lines do not contribute to a main primitive, and are only accessible in a geometry shader.</td>
</tr>
<tr>
<td></td>
<td>Winding Order</td>
<td>The relative order in which vertices are defined within a primitive, used in the facing determination. This ordering has no specific start or end point.</td>
</tr>
</tbody>
</table>

The diagrams are supported with mathematical definitions where the vertices (v) and primitives (p) are numbered starting from 0; v_0 is the first vertex in the provided data and p_0 is the first primitive in the set of primitives defined by the vertices and topology.

The primitive topology is specified by the `VkPipelineInputAssemblyStateCreateInfo::topology` property of the currently active pipeline, if the pipeline was not created with `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` enabled.

Otherwise, the primitive topology is set by calling:

```
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetPrimitiveTopologyEXT(
    VkCommandBuffer   commandBuffer,
    VkPrimitiveTopology primitiveTopology);
```

• **commandBuffer** is the command buffer into which the command will be recorded.

• **topology** specifies the primitive topology to use for drawing.
Valid Usage

- VUID-vkCmdSetPrimitiveTopologyEXT-None-03347
  The extendedDynamicState feature must be enabled

Valid Usage (Implicit)

- VUID-vkCmdSetPrimitiveTopologyEXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle
- VUID-vkCmdSetPrimitiveTopologyEXT-primitiveTopology-parameter
  primitiveTopology must be a valid VkPrimitiveTopology value
- VUID-vkCmdSetPrimitiveTopologyEXT-commandBuffer-recording
  commandBuffer must be in the recording state
- VUID-vkCmdSetPrimitiveTopologyEXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21.1.1. Topology Class

The primitive topologies are grouped into the following topology classes:

Table 30. Topology classes

<table>
<thead>
<tr>
<th>Topology Class</th>
<th>Primitive Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>VK_PRIMITIVE_TOPOLOGY_POINT_LIST</td>
</tr>
<tr>
<td>Topology Class</td>
<td>Primitive Topology</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Line</td>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_LIST, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP, VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY</td>
</tr>
<tr>
<td>Triangle</td>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</td>
</tr>
<tr>
<td>Patch</td>
<td>VK_PRIMITIVE_TOPOLOGY_PATCH_LIST</td>
</tr>
</tbody>
</table>

### 21.1.2. Point Lists

When the topology is `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, each consecutive vertex defines a single point primitive, according to the equation:

\[ p_i = \{v_i\} \]

As there is only one vertex, that vertex is the provoking vertex. The number of primitives generated is equal to \(\text{vertexCount}\).

### 21.1.3. Line Lists

When the topology is `VK_PRIMITIVE_TOPOLOGY_LINE_LIST`, each consecutive pair of vertices defines a single line primitive, according to the equation:

\[ p_i = \{v_{2i}, v_{2i+1}\} \]

The number of primitives generated is equal to \(\lfloor \text{vertexCount}/2 \rfloor\).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \(p_i\) is \(v_{2i}\).
When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{2i+1} \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_i \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{i+1} \).

### 21.1.4. Line Strips

When the topology is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`, one line primitive is defined by each vertex and the following vertex, according to the equation:

\[
p_i = \{v_i, v_{i+1}\}
\]

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-1) \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_i \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{i+1} \).

### 21.1.5. Triangle Lists

When the topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, each consecutive set of three vertices defines a single triangle primitive, according to the equation:

\[
p_i = \{v_{3i}, v_{3i+1}, v_{3i+2}\}
\]

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/3 \rfloor \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{3i} \).
When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{3i+2} \).

### 21.1.6. Triangle Strips

When the topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, one triangle primitive is defined by each vertex and the two vertices that follow it, according to the equation:

\[
p_i = \{ v_i, v_{i+(1+i\%2)}, v_{i+(2-i\%2)} \}
\]

The number of primitives generated is equal to \( \max(0, \text{vertexCount}-2) \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_i \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{i+2} \).

**Note**

The ordering of the vertices in each successive triangle is reversed, so that the winding order is consistent throughout the strip.
21.1.7. Triangle Fans

When the topology is VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN, triangle primitives are defined around a shared common vertex, according to the equation:

\[ p_i = \{v_{i+1}, v_{i+2}, v_0\} \]

The number of primitives generated is equal to \( \text{max}(0, \text{vertexCount}-2) \).

When the provokingVertexMode is VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT, the provoking vertex for \( p_i \) is \( v_{i+1} \).

When the provokingVertexMode is VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT, the provoking vertex for \( p_i \) is \( v_{i+2} \).

Note

If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::triangleFans is VK_FALSE, then triangle fans are not supported by the implementation, and VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN must not be used.

21.1.8. Line Lists With Adjacency

When the topology is VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, each consecutive set of four vertices defines a single line primitive with adjacency, according to the equation:

\[ p_i = \{v_{4i}, v_{4i+1}, v_{4i+2}, v_{4i+3}\} \]

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a geometry shader.

The number of primitives generated is equal to \( \lfloor \text{vertexCount}/4 \rfloor \).

When the provokingVertexMode is VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT, the provoking vertex for \( p_i \) is \( v_{4i+1} \).
When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for $p_i$ is $v_{4i+2}$.

**21.1.9. Line Strips With Adjacency**

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`, one line primitive with adjacency is defined by each vertex and the following vertex, according to the equation:

$$p_i = \{v_i, v_{i+1}, v_{i+2}, v_{i+3}\}$$

A line primitive is described by the second and third vertices of the total primitive, with the remaining two vertices only accessible in a geometry shader.

The number of primitives generated is equal to $\max(0, \text{vertexCount}-3)$.

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for $p_i$ is $v_{i+1}$.

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for $p_i$ is $v_{i+2}$.

**21.1.10. Triangle Lists With Adjacency**

When the `topology` is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY`, each consecutive set of six vertices defines a single triangle primitive with adjacency, according to the equations:

$$p_i = \{v_{6i}, v_{6i+1}, v_{6i+2}, v_{6i+3}, v_{6i+4}, v_{6i+5}\}$$

A triangle primitive is described by the first, third, and fifth vertices of the total primitive, with the remaining three vertices only accessible in a geometry shader.

The number of primitives generated is equal to $\lfloor \text{vertexCount}/6 \rfloor$. 

1302 | Chapter 21. Drawing Commands
When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{6i} \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is \( v_{6i+4} \).

### 21.1.11. Triangle Strips With Adjacency

When the topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`, one triangle primitive with adjacency is defined by each vertex and the following 5 vertices.

The number of primitives generated, \( n \), is equal to \( \lfloor \max(0, \text{vertexCount} - 4)/2 \rfloor \).

If \( n=1 \), the primitive is defined as:

\[ p = \{ v_0, v_1, v_2, v_5, v_4, v_3 \} \]

If \( n>1 \), the total primitive consists of different vertices according to where it is in the strip:

\[ p_i = \{ v_{2i}, v_{2i+1}, v_{2i+2}, v_{2i+6}, v_{2i+4}, v_{2i+3} \} \text{ when } i=0 \]

\[ p_i = \{ v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+6}, v_{2i+2}, v_{2i+1} \} \text{ when } i>0, i<n-1, \text{ and } i\%2=1 \]

\[ p_i = \{ v_{2i}, v_{2i+2}, v_{2i+3}, v_{2i+6}, v_{2i+4}, v_{2i+3} \} \text{ when } i>0, i<n-1, \text{ and } i\%2=0 \]

\[ p_i = \{ v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i+1} \} \text{ when } i=n-1 \text{ and } i\%2=1 \]
\[ p_i = \{v_{2i}, v_{2i+2}, v_{2i+5}, v_{2i+4}, v_{2i+3}\} \text{ when } i=n-1 \text{ and } i\%2=0 \]

A triangle primitive is described by the first, third, and fifth vertices of the total primitive in all cases, with the remaining three vertices only accessible in a geometry shader.

**Note**

The ordering of the vertices in each successive triangle is altered so that the winding order is consistent throughout the strip.

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the provoking vertex for \( p_i \) is always \( v_{2i} \).

When the `provokingVertexMode` is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the provoking vertex for \( p_i \) is always \( v_{2i+4} \).
21.1.12. Patch Lists

When the topology is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, each consecutive set of m vertices defines a single patch primitive, according to the equation:

\[ p_i = \{v_{mi}, v_{mi+1}, \ldots, v_{mi+(m-2)}, v_{mi+(m-1)}\} \]

where m is equal to `VkPipelineTessellationStateCreateInfo::patchControlPoints`.

Patch lists are never passed to vertex post-processing, and as such no provoking vertex is defined for patch primitives. The number of primitives generated is equal to \([vertexCount/m]\).

The vertices comprising a patch have no implied geometry, and are used as inputs to tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

21.2. Primitive Order

Primitives generated by drawing commands progress through the stages of the graphics pipeline in primitive order. Primitive order is initially determined in the following way:

1. Submission order determines the initial ordering
2. For indirect draw commands, the order in which accessed instances of the `VkDrawIndirectCommand` are stored in buffer, from lower indirect buffer addresses to higher addresses.
3. If a draw command includes multiple instances, the order in which instances are executed, from lower numbered instances to higher.

4. The order in which primitives are specified by a draw command:

   ◦ For non-indexed draws, from vertices with a lower numbered $vertexIndex$ to a higher numbered $vertexIndex$.

   ◦ For indexed draws, vertices sourced from a lower index buffer addresses to higher addresses.

   ◦ For draws using mesh shaders, the order is provided by mesh shading.

Within this order implementations further sort primitives:

5. If tessellation shading is active, by an implementation-dependent order of new primitives generated by tessellation.

6. If geometry shading is active, by the order new primitives are generated by geometry shading.

7. If the polygon mode is not $VK_POLYGON_MODE_FILL$, or $VK_POLYGON_MODE_FILL_RECTANGLE_NV$, by an implementation-dependent ordering of the new primitives generated within the original primitive.

Primitive order is later used to define rasterization order, which determines the order in which fragments output results to a framebuffer.

### 21.3. Programmable Primitive Shading

Once primitives are assembled, they proceed to the vertex shading stage of the pipeline. If the draw includes multiple instances, then the set of primitives is sent to the vertex shading stage multiple times, once for each instance.

It is implementation-dependent whether vertex shading occurs on vertices that are discarded as part of incomplete primitives, but if it does occur then it operates as if they were vertices in complete primitives and such invocations can have side effects.

Vertex shading receives two per-vertex inputs from the primitive assembly stage - the $vertexIndex$ and the $instanceIndex$. How these values are generated is defined below, with each command.

Drawing commands fall roughly into two categories:

- Non-indexed drawing commands present a sequential $vertexIndex$ to the vertex shader. The sequential index is generated automatically by the device (see Fixed-Function Vertex Processing for details on both specifying the vertex attributes indexed by $vertexIndex$, as well as binding vertex buffers containing those attributes to a command buffer). These commands are:
  - `vkCmdDraw`
  - `vkCmdDrawIndirect`
  - `vkCmdDrawIndirectCountKHR`
  - `vkCmdDrawIndirectCountAMD`

- Indexed drawing commands read index values from an index buffer and use this to compute the
vertexIndex value for the vertex shader. These commands are:

- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndexedIndirectCountKHR
- vkCmdDrawIndexedIndirectCountAMD

To bind an index buffer to a command buffer, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindIndexBuffer(
    VkCommandBuffer commandBuffer,  // Provided by VK_VERSION_1_0
    VkBuffer buffer,  // Provided by VK_VERSION_1_0
    VkDeviceSize offset,  // Provided by VK_VERSION_1_0
    VkIndexType indexType);  // Provided by VK_VERSION_1_0
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer being bound.
- `offset` is the starting offset in bytes within `buffer` used in index buffer address calculations.
- `indexType` is a `VkIndexType` value specifying whether indices are treated as 16 bits or 32 bits.

### Valid Usage

- VUID-vkCmdBindIndexBuffer-offset-00431
  
  `offset` must be less than the size of `buffer`

- VUID-vkCmdBindIndexBuffer-offset-00432
  
  The sum of `offset` and the address of the range of `VkDeviceMemory` object that is backing `buffer`, must be a multiple of the type indicated by `indexType`

- VUID-vkCmdBindIndexBuffer-buffer-00433
  
  `buffer` must have been created with the `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` flag

- VUID-vkCmdBindIndexBuffer-buffer-00434
  
  If `buffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- VUID-vkCmdBindIndexBuffer-indexType-02507
  
  `indexType` must not be `VK_INDEX_TYPE_NONE_KHR`

- VUID-vkCmdBindIndexBuffer-indexType-02765
  
  If `indexType` is `VK_INDEX_TYPE_UINT8_EXT`, the `indexTypeUint8` feature must be enabled
Valid Usage (Implicit)

- VUID-vkCmdBindIndexBuffer-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdBindIndexBuffer-buffer-parameter
  
  `buffer` must be a valid `VkBuffer` handle

- VUID-vkCmdBindIndexBuffer-indexType-parameter
  
  `indexType` must be a valid `VkIndexType` value

- VUID-vkCmdBindIndexBuffer-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdBindIndexBuffer-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdBindIndexBuffer-commonparent
  
  Both of `buffer`, and `commandBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible values of `vkCmdBindIndexBuffer::indexType`, specifying the size of indices, are:


```c
// Provided by VK_VERSION_1_0
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0,
    VK_INDEX_TYPE_UINT32 = 1,
    // Provided by VK_KHR_acceleration_structure
    VK_INDEX_TYPE_NONE_KHR = 1000165000,
    // Provided by VK_EXT_index_type_uint8
    VK_INDEX_TYPE_UINT8_EXT = 1000265000,
    // Provided by VK_NV_ray_tracing
    VK_INDEX_TYPE_NONE_NV = VK_INDEX_TYPE_NONE_KHR,
    } VkIndexType;
```

- `VK_INDEX_TYPE_UINT16` specifies that indices are 16-bit unsigned integer values.
- `VK_INDEX_TYPE_UINT32` specifies that indices are 32-bit unsigned integer values.
- `VK_INDEX_TYPE_NONE_KHR` specifies that no indices are provided.
- `VK_INDEX_TYPE_UINT8_EXT` specifies that indices are 8-bit unsigned integer values.

The parameters for each drawing command are specified directly in the command or read from buffer memory, depending on the command. Drawing commands that source their parameters from buffer memory are known as indirect drawing commands.

All drawing commands interact with the Robust Buffer Access feature.

To record a non-indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDraw(  
    VkCommandBuffer commandBuffer,  
    uint32_t vertexCount,  
    uint32_t instanceCount,  
    uint32_t firstVertex,  
    uint32_t firstInstance);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `vertexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstVertex` is the index of the first vertex to draw.
- `firstInstance` is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and `vertexCount` consecutive vertex indices with the first `vertexIndex` value equal to `firstVertex`. The primitives are drawn `instanceCount` times with `instanceIndex` starting with `firstInstance` and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.
Valid Usage

- **VUID-vkCmdDraw-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDraw-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDraw-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDraw-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- **VUID-vkCmdDraw-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDraw-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDraw-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- **VUID-vkCmdDraw-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- **VUID-vkCmdDraw-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
Compatibility

- VUID-vkCmdDraw-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdDraw-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdDraw-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdDraw-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdDraw-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdDraw-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdDraw-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdDraw-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDraw-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDraw-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
A **VkPipeline** object bound to the pipeline bind point used by this command **must** not be a protected resource.

- **VUID-vkCmdDraw-None-04115**
  If a **VkImageView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction **must** have at least as many components as the image view’s format.

- **VUID-vkCmdDraw-OpImageWrite-04469**
  If a **VkBufferView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction **must** have at least as many components as the buffer view’s format.

- **VUID-vkCmdDraw-SampledType-04470**
  If a **VkImageView** with a **VkFormat** that has a 64-bit channel width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 64.

- **VUID-vkCmdDraw-SampledType-04471**
  If a **VkImageView** with a **VkFormat** that has a channel width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 32.

- **VUID-vkCmdDraw-SampledType-04472**
  If a **VkBufferView** with a **VkFormat** that has a 64-bit channel width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 64.

- **VUID-vkCmdDraw-SampledType-04473**
  If a **VkBufferView** with a **VkFormat** that has a channel width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 32.

- **VUID-vkCmdDraw-sparseImageInt64Atomics-04474**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkImage** objects created with the **VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT** flag **must** not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdDraw-sparseImageInt64Atomics-04475**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkBuffer** objects created with the **VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT** flag **must** not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdDraw-renderPass-02684**
  The current render pass **must** be compatible with the **renderPass** member of the **VkGraphicsPipelineCreateInfo** structure specified when creating the **VkPipeline** bound to **VK_PIPELINE_BIND_POINT_GRAPHICS**.

- **VUID-vkCmdDraw-subpass-02685**
  The subpass index of the current render pass **must** be equal to the **subpass** member of the **VkGraphicsPipelineCreateInfo** structure specified when creating the **VkPipeline** bound to **VK_PIPELINE_BIND_POINT_GRAPHICS**.

- **VUID-vkCmdDraw-None-02686**
  Every input attachment used by the current subpass **must** be bound to the pipeline via a...
• VUID-vkCmdDraw-None-04584

Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

• VUID-vkCmdDraw-maxMultiviewInstanceIndex-02688

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

• VUID-vkCmdDraw-sampleLocationsEnable-02689

If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set.

• VUID-vkCmdDraw-viewportCount-03417

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

• VUID-vkCmdDraw-scissorCount-03418

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the scissorCount parameter of vkCmdSetScissorWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

• VUID-vkCmdDraw-viewportCount-03419

If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled, then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the scissorCount parameter of vkCmdSetScissorWithCountEXT.

• VUID-vkCmdDraw-viewportCount-04137

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV dynamic state enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDraw-None-04876
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT dynamic state enabled then vkCmdSetRasterizerDiscardEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDraw-None-04877
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT dynamic state enabled then vkCmdSetDepthBiasEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDraw-logicOp-04878
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LOGIC_OP_EXT dynamic state enabled then vkCmdSetLogicOpEXT must have been called in the current command buffer prior to this draw command and the logicOp must be a valid VkLogicOp value

- VUID-vkCmdDraw-None-04879
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT dynamic state enabled then vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDraw-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the PrimitiveShadingRateKHR built-in, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must be 1

- VUID-vkCmdDraw-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDraw-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDraw-pStrides-04884
  If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of...
vkCmdBindVertexBuffers2EXT must not be NULL

- VUID-vkCmdDraw-commandBuffer-02712
  If commandBuffer is a protected command buffer, any resource written to by the VkPipeline object bound to the pipeline bind point used by this command must not be an unprotected resource

- VUID-vkCmdDraw-commandBuffer-02713
  If commandBuffer is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the VkPipeline object bound to the pipeline bind point used by this command must not write to any resource

- VUID-vkCmdDraw-commandBuffer-04617
  If any of the shader stages of the VkPipeline bound to the pipeline bind point used by this command uses the RayQueryKHR capability, then commandBuffer must not be a protected command buffer

- VUID-vkCmdDraw-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface must have either valid or VK_NULL_HANDLE buffers bound

- VUID-vkCmdDraw-None-04008
  If the nullDescriptor feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface must not be VK_NULL_HANDLE

- VUID-vkCmdDraw-None-02721
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description

Valid Usage (Implicit)

- VUID-vkCmdDraw-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDraw-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDraw-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDraw-renderpass
  This command must only be called inside of a render pass instance
Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indexed draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexed(  
  VkCommandBuffer commandBuffer,  
  uint32_t indexCount,  
  uint32_t instanceCount,  
  uint32_t firstIndex,  
  int32_t vertexOffset,  
  uint32_t firstInstance);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `indexCount` is the number of vertices to draw.
- `instanceCount` is the number of instances to draw.
- `firstIndex` is the base index within the index buffer.
- `vertexOffset` is the value added to the vertex index before indexing into the vertex buffer.
- `firstInstance` is the instance ID of the first instance to draw.

When the command is executed, primitives are assembled using the current primitive topology and `indexCount` vertices whose indices are retrieved from the index buffer. The index buffer is treated as an array of tightly packed unsigned integers of size defined by the `vkCmdBindIndexBuffer ::indexType` parameter with which the buffer was bound.

The first vertex index is at an offset of `firstIndex × indexSize + offset` within the bound index buffer, where `offset` is the offset specified by `vkCmdBindIndexBuffer` and `indexSize` is the byte size of the type specified by `indexType`. Subsequent index values are retrieved from consecutive locations in the index buffer. Indices are first compared to the primitive restart value, then zero extended to 32 bits (if the `indexType` is `VK_INDEX_TYPE_UINT8_EXT` or `VK_INDEX_TYPE_UINT16`) and have `vertexOffset` added to them, before being supplied as the `vertexIndex` value.
The primitives are drawn $\text{instanceCount}$ times with $\text{instanceIndex}$ starting with $\text{firstInstance}$ and increasing sequentially for each instance. The assembled primitives execute the bound graphics pipeline.
Valid Usage

- VUID-vkCmdDrawIndexed-magFilter-04553
  If a VkSampler created with magFilter or minFilter equal to VK_FILTER_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-vkCmdDrawIndexed-mipmapMode-04770
  If a VkSampler created with mipmapMode equal to VK_SAMPLER_MIPMAP_MODE_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-vkCmdDrawIndexed-None-02691
  If a VkImageView is accessed using atomic operations as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT

- VUID-vkCmdDrawIndexed-None-02692
  If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

- VUID-vkCmdDrawIndexed-filterCubic-02694
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2

- VUID-vkCmdDrawIndexed-filterCubicMinmax-02695
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

- VUID-vkCmdDrawIndexed-flags-02696
  Any VkImage created with a VkImageCreateInfo::flags containing VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV sampled as a result of this command must only be sampled using a VkSamplerAddressMode of VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE

- VUID-vkCmdDrawIndexed-None-02697
  For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDrawIndexed-None-02698
  For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout
Compatibility

- VUID-vkCmdDrawIndexed-None-02699
  Descriptors in each bound descriptor set, specified via \texttt{vkCmdBindDescriptorSets}, \textbf{must} be valid if they are statically used by the \texttt{VkPipeline} bound to the pipeline bind point used by this command

- VUID-vkCmdDrawIndexed-None-02700
  A valid pipeline \textbf{must} be bound to the pipeline bind point used by this command

- VUID-vkCmdDrawIndexed-commandBuffer-02701
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command requires any dynamic state, that state \textbf{must} have been set or inherited for \texttt{commandBuffer}, and done so after any previously bound pipeline with the corresponding state not specified as dynamic

- VUID-vkCmdDrawIndexed-None-02859
  There \textbf{must} not have been any calls to dynamic state setting commands for any state not specified as dynamic in the \texttt{VkPipeline} object bound to the pipeline bind point used by this command, since that pipeline was bound

- VUID-vkCmdDrawIndexed-None-02702
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a \texttt{VkSampler} object that uses unnormalized coordinates, that sampler \textbf{must} not be used to sample from any \texttt{VkImage} with a \texttt{VkImageView} of the type \texttt{VK_IMAGE_VIEW_TYPE_3D}, \texttt{VK_IMAGE_VIEW_TYPE_CUBE}, \texttt{VK_IMAGE_VIEW_TYPE_1D_ARRAY}, \texttt{VK_IMAGE_VIEW_TYPE_2D_ARRAY} or \texttt{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY}, in any shader stage

- VUID-vkCmdDrawIndexed-None-02703
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a \texttt{VkSampler} object that uses unnormalized coordinates, that sampler \textbf{must} not be used with any of the SPIR-V \texttt{OpImageSample*} or \texttt{OpImageSparseSample*} instructions with \texttt{ImplicitLod}, \texttt{Dref} or \texttt{Proj} in their name, in any shader stage

- VUID-vkCmdDrawIndexed-None-02704
  If the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a \texttt{VkSampler} object that uses unnormalized coordinates, that sampler \textbf{must} not be used with any of the SPIR-V \texttt{OpImageSample*} or \texttt{OpImageSparseSample*} instructions that includes a LOD bias or any offset values, in any shader stage

- VUID-vkCmdDrawIndexed-None-02705
  If the \texttt{robust buffer access} feature is not enabled, and if the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a uniform buffer, it \textbf{must} not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- VUID-vkCmdDrawIndexed-None-02706
  If the \texttt{robust buffer access} feature is not enabled, and if the \texttt{VkPipeline} object bound to the pipeline bind point used by this command accesses a storage buffer, it \textbf{must} not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point

- VUID-vkCmdDrawIndexed-commandBuffer-02707
  If \texttt{commandBuffer} is an unprotected command buffer, any resource accessed by the
 VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- VUID-vkCmdDrawIndexed-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format.

- VUID-vkCmdDrawIndexed-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view's format.

- VUID-vkCmdDrawIndexed-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndexed-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndexed-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndexed-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndexed-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndexed-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndexed-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawIndexed-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawIndexed-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a...
VUID-vkCmdDrawIndexed-None-04584
Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

VUID-vkCmdDrawIndexed-maxMultiviewInstanceIndex-02688
If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

VUID-vkCmdDrawIndexed-sampleLocationsEnable-02689
If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set.

VUID-vkCmdDrawIndexed-viewportCount-03417
If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

VUID-vkCmdDrawIndexed-scissorCount-03418
If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the scissorCount parameter of vkCmdSetScissorWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

VUID-vkCmdDrawIndexed-viewportCount-03419
If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the scissorCount parameter of vkCmdSetScissorWithCountEXT.

VUID-vkCmdDrawIndexed-viewportCount-04137
If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV dynamic state enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV`::`viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV`::`viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV`::`exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo`::`topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then
vkCmdSetPatchControlPointsEXT **must** have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexed-None-04876**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT **must**` have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexed-None-04877**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT **must**` have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexed-logicOp-04878**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT **must**` have been called in the current command buffer prior to this draw command and the `logicOp **must**` be a valid `VkLogicOp` value

- **VUID-vkCmdDrawIndexed-None-04879**
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT **must**` have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawIndexed-primitiveFragmentShadingRateWithMultipleViewports-04552**
  If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT **must**` have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT **must**` be `1`

- **VUID-vkCmdDrawIndexed-blendEnable-04727**
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's `format features` do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState **must**` be `VK_FALSE`

- **VUID-vkCmdDrawIndexed-rasterizationSamples-04740**
  If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples **must**` be the same as the current subpass color and/or depth/stencil attachments

- **VUID-vkCmdDrawIndexed-pStrides-04884**
  If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT **must**` have been called in the current command buffer prior to this draw command, and the `pStrides` parameter of
**vkCmdBindVertexBuffers2EXT** **must** not be **NULL**

- VUID-vkCmdDrawIndexed-commandBuffer-02712
  If `commandBuffer` is a protected command buffer, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command **must** not be an unprotected resource

- VUID-vkCmdDrawIndexed-commandBuffer-02713
  If `commandBuffer` is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point used by this command **must** not write to any resource

- VUID-vkCmdDrawIndexed-commandBuffer-04617
  If any of the shader stages of the `VkPipeline` bound to the pipeline bind point used by this command uses the `RayQueryKHR` capability, then `commandBuffer` **must** not be a protected command buffer

- VUID-vkCmdDrawIndexed-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** have either valid or `VK_NULL_HANDLE` buffers bound

- VUID-vkCmdDrawIndexed-None-04008
  If the `nullDescriptor` feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface **must** not be `VK_NULL_HANDLE`

- VUID-vkCmdDrawIndexed-None-02721
  For a given vertex buffer binding, any attribute data fetched **must** be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description

- VUID-vkCmdDrawIndexed-indexSize-00463
  \((\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\) **must** be less than or equal to the size of the bound index buffer, with `indexSize` being based on the type specified by `indexType`, where the index buffer, `indexType`, and `offset` are specified via `vkCmdBindIndexBuffer`

---

**Valid Usage (Implicit)**

- VUID-vkCmdDrawIndexed-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdDrawIndexed-commandBuffer-recording
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdDrawIndexed-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

- VUID-vkCmdDrawIndexed-renderpass
  This command **must** only be called inside of a render pass instance
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record a non-indexed indirect draw, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDrawIndirect(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    uint32_t drawCount,
    uint32_t stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `drawCount` is the number of draws to execute, and can be zero.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirect` behaves similarly to `vkCmdDraw` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawIndirectCommand` structures. If `drawCount` is less than or equal to one, `stride` is ignored.
Valid Usage

• VUID-vkCmdDrawIndirect-magFilter-04553
If a VkSampler created with magFilter or minFilter equal to VK_FILTER_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

• VUID-vkCmdDrawIndirect-mipmapMode-04770
If a VkSampler created with.mipmapMode equal to VK_SAMPLER_MIPMAP_MODE_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

• VUID-vkCmdDrawIndirect-None-02691
If a VkImageView is accessed using atomic operations as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT

• VUID-vkCmdDrawIndirect-None-02692
If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

• VUID-vkCmdDrawIndirect-filterCubic-02694
Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2

• VUID-vkCmdDrawIndirect-filterCubicMinmax-02695
Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

• VUID-vkCmdDrawIndirect-flags-02696
Any VkImage created with a VkImageCreateInfo::flags containing VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV sampled as a result of this command must only be sampled using a VkSamplerAddressMode of VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE

• VUID-vkCmdDrawIndirect-None-02697
For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

• VUID-vkCmdDrawIndirect-None-02698
For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout
Compatibility

- VUID-vkCmdDrawIndirect-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdDrawIndirect-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdDrawIndirect-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdDrawIndirect-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdDrawIndirect-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdDrawIndirect-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdDrawIndirect-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdDrawIndirect-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDrawIndirect-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDrawIndirect-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
 VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

• VUID-vkCmdDrawIndirect-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

• VUID-vkCmdDrawIndirect-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

• VUID-vkCmdDrawIndirect-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

• VUID-vkCmdDrawIndirect-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

• VUID-vkCmdDrawIndirect-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

• VUID-vkCmdDrawIndirect-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

• VUID-vkCmdDrawIndirect-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

• VUID-vkCmdDrawIndirect-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

• VUID-vkCmdDrawIndirect-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-vkCmdDrawIndirect-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-vkCmdDrawIndirect-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a
descriptor set

- **VUID-vkCmdDrawIndirect-None-04584**
  Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

- **VUID-vkCmdDrawIndirect-maxMultiviewInstanceIndex-02688**
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

- **VUID-vkCmdDrawIndirect-sampleLocationsEnable-02689**
  If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set.

- **VUID-vkCmdDrawIndirect-viewportCount-03417**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

- **VUID-vkCmdDrawIndirect-scissorCount-03418**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the scissorCount parameter of vkCmdSetScissorWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

- **VUID-vkCmdDrawIndirect-viewportCount-03419**
  If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled, then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the scissorCount parameter of vkCmdSetScissorWithCountEXT.

- **VUID-vkCmdDrawIndirect-viewportCount-04137**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV dynamic state enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT.
- VUID-vkCmdDrawIndirect-viewportCount-04138
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

- VUID-vkCmdDrawIndirect-viewportCount-04139
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV`::`viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

- VUID-vkCmdDrawIndirect-viewportCount-04140
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

- VUID-vkCmdDrawIndirect-VkPipelineVieportCreateInfo-04141
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV`::`viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

- VUID-vkCmdDrawIndirect-VkPipelineVieportCreateInfo-04142
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV`::`exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

- VUID-vkCmdDrawIndirect-primitiveTopology-03420
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo`::`topology` state.

- VUID-vkCmdDrawIndirect-None-04875
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then
**vkCmdSetPatchControlPoints** must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirect-None-04876
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT** dynamic state enabled then **vkCmdSetRasterizerDiscardEnable** must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirect-None-04877
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT** dynamic state enabled then **vkCmdSetDepthBiasEnable** must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirect-logicOp-04878
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_LOGIC_OP_EXT** dynamic state enabled then **vkCmdSetLogicOp** must have been called in the current command buffer prior to this draw command and the **logicOp** must be a valid **VkLogicOp** value

- VUID-vkCmdDrawIndirect-None-04879
  If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT** dynamic state enabled then **vkCmdSetPrimitiveRestartEnable** must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirect-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the **primitiveFragmentShadingRateWithMultipleViewports** limit is not supported, the bound graphics pipeline was created with the **VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT** dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the **PrimitiveShadingRateKHR** built-in, then **vkCmdSetViewportWithCount** must have been called in the current command buffer prior to this draw command, and the **viewportCount** parameter of **vkCmdSetViewportWithCount** must be 1

- VUID-vkCmdDrawIndirect-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's **format features** do not contain **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT**, then the **blendEnable** member of the corresponding element of the **pAttachments** member of **pColorBlendState** must be **VK_FALSE**

- VUID-vkCmdDrawIndirect-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the **VK_AMD_mixed_attachment_samples** nor the **VK_NV_framebuffer_mixed_samples** extensions are enabled, then **VkPipelineMultisampleStateCreateInfo::rasterizationSamples** must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndirect-pStrides-04884
  If the bound graphics pipeline was created with the **VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT** dynamic state enabled, then **vkCmdBindVertexBuffers2** must have been called in the current command buffer prior to this draw command, and the **pStrides** parameter of
**vkCmdBindVertexBuffers2EXT** must not be **NULL**

- VUID-vkCmdDrawIndirect-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or **VK_NULL_HANDLE** buffers bound

- VUID-vkCmdDrawIndirect-None-04008
  If the **nullDescriptor** feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be **VK_NULL_HANDLE**

- VUID-vkCmdDrawIndirect-None-02721
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in **Vertex Input Description**

- VUID-vkCmdDrawIndirect-buffer-02708
  If **buffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

- VUID-vkCmdDrawIndirect-buffer-02709
  **buffer** must have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set

- VUID-vkCmdDrawIndirect-offset-02710
  **offset** must be a multiple of 4

- VUID-vkCmdDrawIndirect-commandBuffer-02711
  **commandBuffer** must not be a protected command buffer

- VUID-vkCmdDrawIndirect-drawCount-02718
  If the **multi-draw indirect** feature is not enabled, **drawCount** must be 0 or 1

- VUID-vkCmdDrawIndirect-drawCount-02719
  **drawCount** must be less than or equal to **VkPhysicalDeviceLimits::maxDrawIndirectCount**

- VUID-vkCmdDrawIndirect-firstInstance-00478
  If the **drawIndirectFirstInstance** feature is not enabled, all the **firstInstance** members of the **VkDrawIndirectCommand** structures accessed by this command must be 0

- VUID-vkCmdDrawIndirect-drawCount-00476
  If **drawCount** is greater than 1, **stride** must be a multiple of 4 and **must** be greater than or equal to sizeof(**VkDrawIndirectCommand**)  

- VUID-vkCmdDrawIndirect-drawCount-00477
  If **drawCount** is equal to 1, (offset + sizeof(**VkDrawIndirectCommand**)) **must** be less than or equal to the size of **buffer**  

- VUID-vkCmdDrawIndirect-drawCount-00487
  If **drawCount** is equal to 1, (stride × (drawCount - 1) + offset + sizeof(**VkDrawIndirectCommand**)) **must** be less than or equal to the size of **buffer**
Valid Usage (Implicit)

- VUID-vkCmdDrawIndirect-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDrawIndirect-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-vkCmdDrawIndirect-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDrawIndirect-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDrawIndirect-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdDrawIndirect-commonparent
  Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The VkDrawIndirectCommand structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDrawIndirectCommand {
    uint32_t vertexCount;
    uint32_t instanceCount;
    uint32_t firstVertex;
    uint32_t firstInstance;
} VkDrawIndirectCommand;
```

- vertexCount is the number of vertices to draw.
• instanceCount is the number of instances to draw.
• firstVertex is the index of the first vertex to draw.
• firstInstance is the instance ID of the first instance to draw.

The members of VkDrawIndirectCommand have the same meaning as the similarly named parameters of vkCmdDraw.

Valid Usage
• VUID-VkDrawIndirectCommand-None-00500
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description
• VUID-VkDrawIndirectCommand-firstInstance-00501
  If the drawIndirectFirstInstance feature is not enabled, firstInstance must be 0

To record a non-indexed draw call with a draw call count sourced from a buffer, call:

```c
// Provided by VK_KHR_draw_indirect_count
void vkCmdDrawIndirectCountKHR(
  VkCommandBuffer                        commandBuffer,
  VkBuffer                               buffer,
  VkDeviceSize                           offset,
  VkBuffer                               countBuffer,
  VkDeviceSize                           countBufferOffset,
  uint32_t                               maxDrawCount,
  uint32_t                               stride);
```
or the equivalent command

```c
// Provided by VK_AMD_draw_indirect_count
void vkCmdDrawIndirectCountAMD(
  VkCommandBuffer                        commandBuffer,
  VkBuffer                               buffer,
  VkDeviceSize                           offset,
  VkBuffer                               countBuffer,
  VkDeviceSize                           countBufferOffset,
  uint32_t                               maxDrawCount,
  uint32_t                               stride);
```

• commandBuffer is the command buffer into which the command is recorded.
• buffer is the buffer containing draw parameters.
• offset is the byte offset into buffer where parameters begin.
• countBuffer is the buffer containing the draw count.
• countBufferOffset is the byte offset into countBuffer where the draw count begins.
- `maxDrawCount` specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in `countBuffer` and `maxDrawCount`.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawIndirectCount` behaves similarly to `vkCmdDrawIndirect` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from `countBuffer` located at `countBufferOffset` and use this as the draw count.
Valid Usage

- VUID-vkCmdDrawIndirectCount-magFilter-04553
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- VUID-vkCmdDrawIndirectCount-mipmapMode-04770
  If a `VkSampler` created with `.mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- VUID-vkCmdDrawIndirectCount-None-02691
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

- VUID-vkCmdDrawIndirectCount-None-02692
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- VUID-vkCmdDrawIndirectCount-filterCubic-02694
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- VUID-vkCmdDrawIndirectCount-filterCubicMinmax-02695
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- VUID-vkCmdDrawIndirectCount-flags-02696
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

- VUID-vkCmdDrawIndirectCount-None-02697
  For each set $n$ that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to $n$ at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set $n$, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- VUID-vkCmdDrawIndirectCount-None-02698
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.
Compatibility

- VUID-vkCmdDrawIndirectCount-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdDrawIndirectCount-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdDrawIndirectCount-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdDrawIndirectCount-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdDrawIndirectCount-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdDrawIndirectCount-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdDrawIndirectCount-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdDrawIndirectCount-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDrawIndirectCount-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDrawIndirectCount-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
A VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

- VUID-vkCmdDrawIndirectCount-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- VUID-vkCmdDrawIndirectCount-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- VUID-vkCmdDrawIndirectCount-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndirectCount-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndirectCount-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndirectCount-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndirectCount-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndirectCount-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndirectCount-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawIndirectCount-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawIndirectCount-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a...
**descriptor set**

- VUID-vkCmdDrawIndirectCount-None-04584
  Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

- VUID-vkCmdDrawIndirectCount-maxMultiviewInstanceIndex-02688
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`

- VUID-vkCmdDrawIndirectCount-sampleLocationsEnable-02689
  If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

- VUID-vkCmdDrawIndirectCount-viewportCount-03417
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- VUID-vkCmdDrawIndirectCount-scissorCount-03418
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- VUID-vkCmdDrawIndirectCount-viewportCount-03419
  If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

- VUID-vkCmdDrawIndirectCount-viewportCount-04137
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled, then the bound graphics pipeline **must** have been created with `VkPipelineViewportWScalingStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

1340 | Chapter 21. Drawing Commands
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectCount-None-04876
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT` dynamic state enabled then `vkCmdSetRasterizerDiscardEnableEXT` must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectCount-None-04877
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` dynamic state enabled then `vkCmdSetDepthBiasEnableEXT` must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectCount-logicOp-04878
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_LOGIC_OP_EXT` dynamic state enabled then `vkCmdSetLogicOpEXT` must have been called in the current command buffer prior to this draw command and the `logicOp` must be a valid `VkLogicOp` value

- VUID-vkCmdDrawIndirectCount-None-04879
  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT` dynamic state enabled then `vkCmdSetPrimitiveRestartEnableEXT` must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectCount-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the `primitiveFragmentShadingRateWithMultipleViewports` limit is not supported, the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the `PrimitiveShadingRateKHR` built-in, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must be 1

- VUID-vkCmdDrawIndirectCount-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's `format features` do not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`, then the `blendEnable` member of the corresponding element of the `pAttachments` member of `pColorBlendState` must be `VK_FALSE`

- VUID-vkCmdDrawIndirectCount-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the `VK_AMD_mixed_attachment_samples` nor the `VK_NV_framebuffer_mixed_samples` extensions are enabled, then `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndirectCount-pStrides-04884
  If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled, then `vkCmdBindVertexBuffers2EXT` must have been called in the current command buffer prior to this draw command, and the `pStrides` parameter of
**vkCmdBindVertexBuffers2EXT** must not be **NULL**

- **VUID-vkCmdDrawIndirectCount-None-04007**
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or **VK_NULL_HANDLE** buffers bound.

- **VUID-vkCmdDrawIndirectCount-None-04008**
  If the **nullDescriptor** feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be **VK_NULL_HANDLE**.

- **VUID-vkCmdDrawIndirectCount-None-02721**
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in **Vertex Input Description**.

- **VUID-vkCmdDrawIndirectCount-buffer-02708**
  If **buffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object.

- **VUID-vkCmdDrawIndirectCount-buffer-02709**
  **buffer** must have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set.

- **VUID-vkCmdDrawIndirectCount-offset-02710**
  **offset** must be a multiple of 4.

- **VUID-vkCmdDrawIndirectCount-commandBuffer-02711**
  **commandBuffer** must not be a protected command buffer.

- **VUID-vkCmdDrawIndirectCount-countBuffer-02714**
  If **countBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object.

- **VUID-vkCmdDrawIndirectCount-countBuffer-02715**
  **countBuffer** must have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set.

- **VUID-vkCmdDrawIndirectCount-countBufferOffset-02716**
  **countBufferOffset** must be a multiple of 4.

- **VUID-vkCmdDrawIndirectCount-countBuffer-02717**
  The count stored in **countBuffer** must be less than or equal to **VkPhysicalDeviceLimits::maxDrawIndirectCount**.

- **VUID-vkCmdDrawIndirectCount-countBufferOffset-04129**
  (**countBufferOffset** + **sizeof(uint32_t)**) must be less than or equal to the size of **countBuffer**.

- **VUID-vkCmdDrawIndirectCount-stride-03110**
  **stride** must be a multiple of 4 and must be greater than or equal to **sizeof(VkDrawIndirectCommand)**.

- **VUID-vkCmdDrawIndirectCount-maxDrawCount-03111**
  If **maxDrawCount** is greater than or equal to 1, (**stride** × (**maxDrawCount** - 1) + **offset** + **sizeof(VkDrawIndirectCommand)**) must be less than or equal to the size of **buffer**.

- **VUID-vkCmdDrawIndirectCount-countBuffer-03121**
  If the count stored in **countBuffer** is equal to 1, (**offset** + **sizeof(VkDrawIndirectCommand)**)
**must** be less than or equal to the size of *buffer*

- **VUID-vkCmdDrawIndirectCount-countBuffer-03122**
  If the count stored in *countBuffer* is greater than 1, *(stride × (drawCount - 1) + offset + sizeof(VkDrawIndirectCommand)) must** be less than or equal to the size of *buffer*

---

**Valid Usage (Implicit)**

- **VUID-vkCmdDrawIndirectCount-commandBuffer-parameter**
  *commandBuffer* **must** be a valid *VkCommandBuffer* handle
- **VUID-vkCmdDrawIndirectCount-buffer-parameter**
  *buffer** **must** be a valid *VkBuffer* handle
- **VUID-vkCmdDrawIndirectCount-countBuffer-parameter**
  *countBuffer** **must** be a valid *VkBuffer* handle
- **VUID-vkCmdDrawIndirectCount-commandBuffer-recording**
  *commandBuffer** **must** be in the **recording state**
- **VUID-vkCmdDrawIndirectCount-commandBuffer-cmdpool**
  The *VkCommandPool* that *commandBuffer* was allocated from **must** support graphics operations
- **VUID-vkCmdDrawIndirectCount-renderpass**
  This command **must** only be called inside of a render pass instance
- **VUID-vkCmdDrawIndirectCount-commonparent**
  Each of *buffer*, *commandBuffer*, and *countBuffer** **must** have been created, allocated, or retrieved from the same *VkDevice*

---

**Host Synchronization**

- Host access to *commandBuffer** **must** be externally synchronized
- Host access to the *VkCommandPool* that *commandBuffer* was allocated from **must** be externally synchronized

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indexed indirect draw, call:
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexedIndirect(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset,
    uint32_t drawCount,
    uint32_t stride);

- **commandBuffer** is the command buffer into which the command is recorded.
- **buffer** is the buffer containing draw parameters.
- **offset** is the byte offset into **buffer** where parameters begin.
- **drawCount** is the number of draws to execute, and **can** be zero.
- **stride** is the byte stride between successive sets of draw parameters.

**vkCmdDrawIndexedIndirect** behaves similarly to **vkCmdDrawIndexed** except that the parameters are read by the device from a buffer during execution. **drawCount** draws are executed by the command, with parameters taken from **buffer** starting at **offset** and increasing by **stride** bytes for each successive draw. The parameters of each draw are encoded in an array of **VkDrawIndexedIndirectCommand** structures. If **drawCount** is less than or equal to one, **stride** is ignored.
Valid Usage

- VUID-vkCmdDrawIndexedIndirect-magFilter-04553
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- VUID-vkCmdDrawIndexedIndirect-mipmapMode-04770
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- VUID-vkCmdDrawIndexedIndirect-None-02691
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- VUID-vkCmdDrawIndexedIndirect-None-02692
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- VUID-vkCmdDrawIndexedIndirect-filterCubic-02694
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- VUID-vkCmdDrawIndexedIndirect-filterCubicMinmax-02695
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- VUID-vkCmdDrawIndexedIndirect-flags-02696
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- VUID-vkCmdDrawIndexedIndirect-None-02697
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- VUID-vkCmdDrawIndexedIndirect-None-02698
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
Compatibility

• VUID-vkCmdDrawIndexedIndirect-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

• VUID-vkCmdDrawIndexedIndirect-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

• VUID-vkCmdDrawIndexedIndirect-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

• VUID-vkCmdDrawIndexedIndirect-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

• VUID-vkCmdDrawIndexedIndirect-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

• VUID-vkCmdDrawIndexedIndirect-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

• VUID-vkCmdDrawIndexedIndirect-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

• VUID-vkCmdDrawIndexedIndirect-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

• VUID-vkCmdDrawIndexedIndirect-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

• VUID-vkCmdDrawIndexedIndirect-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
 VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- VUID-vkCmdDrawIndexedIndirect-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format.

- VUID-vkCmdDrawIndexedIndirect-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view's format.

- VUID-vkCmdDrawIndexedIndirect-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndexedIndirect-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndexedIndirect-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndexedIndirect-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndexedIndirect-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndexedIndirect-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndexedIndirect-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-vkCmdDrawIndexedIndirect-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-vkCmdDrawIndexedIndirect-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a
descriptor set

- **VUID-vkCmdDrawIndexedIndirect-None-04584**
  
  Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

- **VUID-vkCmdDrawIndexedIndirect-maxMultiviewInstanceIndex-02688**

  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`.

- **VUID-vkCmdDrawIndexedIndirect-sampleLocationsEnable-02689**

  If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment must have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

- **VUID-vkCmdDrawIndexedIndirect-viewportCount-03417**

  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` must have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- **VUID-vkCmdDrawIndexedIndirect-scissorCount-03418**

  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` must have been called in the current command buffer prior to this draw command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` must match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

- **VUID-vkCmdDrawIndexedIndirect-viewportCount-03419**

  If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled, then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` must have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` must match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

- **VUID-vkCmdDrawIndexedIndirect-viewportCount-04137**

  If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportWScalingStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then...
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirect-None-04876
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT dynamic state enabled then
  vkCmdSetRasterizerDiscardEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirect-None-04877
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT dynamic state enabled then
  vkCmdSetDepthBiasEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirect-logicOp-04878
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_LOGIC_OP_EXT dynamic state enabled then
  vkCmdSetLogicOpEXT must have been called in the current command buffer prior to this draw command and the logicOp must be a valid VkLogicOp value

- VUID-vkCmdDrawIndexedIndirect-None-04879
  If the bound graphics pipeline state was created with the
  VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT dynamic state enabled then
  vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirect-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, the
  bound graphics pipeline was created with the
  VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the PrimitiveShadingRateKHR built-in, then
  vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must be 1

- VUID-vkCmdDrawIndexedIndirect-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDrawIndexedIndirect-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the
  VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then
  VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndexedIndirect-pStrides-04884
  If the bound graphics pipeline was created with the
  VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then
  vkCmdBindVertexBuffer2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of
vkCmdBindVertexBuffers2EXT must not be NULL

- VUID-vkCmdDrawIndexedIndirect-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or VK_NULL_HANDLE buffers bound

- VUID-vkCmdDrawIndexedIndirect-None-04008
  If the nullDescriptor feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be VK_NULL_HANDLE

- VUID-vkCmdDrawIndexedIndirect-None-02721
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description

- VUID-vkCmdDrawIndexedIndirect-buffer-02708
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdDrawIndexedIndirect-buffer-02709
  buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-vkCmdDrawIndexedIndirect-offset-02710
  offset must be a multiple of 4

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-02711
  commandBuffer must not be a protected command buffer

- VUID-vkCmdDrawIndexedIndirect-drawCount-02718
  If the multi-draw indirect feature is not enabled, drawCount must be 0 or 1

- VUID-vkCmdDrawIndexedIndirect-drawCount-02719
  drawCount must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount

- VUID-vkCmdDrawIndexedIndirect-drawCount-00528
  If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndexedIndirectCommand)

- VUID-vkCmdDrawIndexedIndirect-firstInstance-00530
  If the drawIndirectFirstInstance feature is not enabled, all the firstInstance members of the VkDrawIndexedIndirectCommand structures accessed by this command must be 0

- VUID-vkCmdDrawIndexedIndirect-drawCount-00539
  If drawCount is equal to 1, (offset + sizeof(VkDrawIndexedIndirectCommand)) must be less than or equal to the size of buffer

- VUID-vkCmdDrawIndexedIndirect-drawCount-00540
  If drawCount is greater than 1, (stride × (drawCount - 1) + offset + sizeof(VkDrawIndexedIndirectCommand)) must be less than or equal to the size of buffer
Valid Usage (Implicit)

- VUID-vkCmdDrawIndexedIndirect-commandBuffer-parameter
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdDrawIndexedIndirect-buffer-parameter
  - `buffer` must be a valid `VkBuffer` handle
- VUID-vkCmdDrawIndexedIndirect-commandBuffer-recording
  - `commandBuffer` must be in the recording state
- VUID-vkCmdDrawIndexedIndirect-commandBuffer-cmdpool
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- VUID-vkCmdDrawIndexedIndirect-renderpass
  - This command must only be called inside of a render pass instance
- VUID-vkCmdDrawIndexedIndirect-commonparent
  - Both of `buffer`, and `commandBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDrawIndexedIndirectCommand` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkDrawIndexedIndirectCommand {
    uint32_t    indexCount;
    uint32_t    instanceCount;
    uint32_t    firstIndex;
    int32_t     vertexOffset;
    uint32_t    firstInstance;
} VkDrawIndexedIndirectCommand;
```
• **indexCount** is the number of vertices to draw.
• **instanceCount** is the number of instances to draw.
• **firstIndex** is the base index within the index buffer.
• **vertexOffset** is the value added to the vertex index before indexing into the vertex buffer.
• **firstInstance** is the instance ID of the first instance to draw.

The members of **VkDrawIndexedIndirectCommand** have the same meaning as the similarly named parameters of **vkCmdDrawIndexed**.

### Valid Usage

- **VUID-VkDrawIndexedIndirectCommand-None-00552**
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in **Vertex Input Description**

- **VUID-VkDrawIndexedIndirectCommand-indexSize-00553**
  \[(\text{indexSize} \times (\text{firstIndex} + \text{indexCount}) + \text{offset})\] must be less than or equal to the size of the bound index buffer, with indexSize being based on the type specified by **indexType**, where the index buffer, **indexType**, and offset are specified via **vkCmdBindIndexBuffer**

- **VUID-VkDrawIndexedIndirectCommand-firstInstance-00554**
  If the **drawIndirectFirstInstance** feature is not enabled, **firstInstance** must be 0

To record an indexed draw call with a draw call count sourced from a buffer, call:

```c
// Provided by VK_KHR_draw_indirect_count
void vkCmdDrawIndexedIndirectCountKHR(
    VkCommandBuffer commandBuffer, commandBuffer,
    VkBuffer buffer, buffer,
    VkDeviceSize offset, offset,
    VkBuffer countBuffer, countBuffer,
    VkDeviceSize countBufferOffset, countBufferOffset,
    uint32_t maxDrawCount, maxDrawCount,
    uint32_t stride); stride);
```

or the equivalent command

```c
// Provided by VK_AMD_draw_indirect_count
void vkCmdDrawIndexedIndirectCountAMD(
    VkCommandBuffer commandBuffer, commandBuffer,
    VkBuffer buffer, buffer,
    VkDeviceSize offset, offset,
    VkBuffer countBuffer, countBuffer,
    VkDeviceSize countBufferOffset, countBufferOffset,
    uint32_t maxDrawCount, maxDrawCount,
    uint32_t stride); stride);
```
• **commandBuffer** is the command buffer into which the command is recorded.

• **buffer** is the buffer containing draw parameters.

• **offset** is the byte offset into **buffer** where parameters begin.

• **countBuffer** is the buffer containing the draw count.

• **countBufferOffset** is the byte offset into **countBuffer** where the draw count begins.

• **maxDrawCount** specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in **countBuffer** and **maxDrawCount**.

• **stride** is the byte stride between successive sets of draw parameters.

**vkCmdDrawIndexedIndirectCount** behaves similarly to **vkCmdDrawIndexedIndirect** except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from **countBuffer** located at **countBufferOffset** and use this as the draw count.
Valid Usage

• **VUID-vkCmdDrawIndexedIndirectCount-magFilter-04553**
  If a **VkSampler** created with **magFilter** or **minFilter** equal to **VK_FILTER_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view's **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**.

• **VUID-vkCmdDrawIndexedIndirectCount-mipmapMode-04770**
  If a **VkSampler** created with **mipmapMode** equal to **VK_SAMPLER_MIPMAP_MODE_LINEAR** and **compareEnable** equal to **VK_FALSE** is used to sample a **VkImageView** as a result of this command, then the image view's **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**.

• **VUID-vkCmdDrawIndexedIndirectCount-None-02691**
  If a **VkImageView** is accessed using atomic operations as a result of this command, then the image view's **format features** **must** contain **VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT**.

• **VUID-vkCmdDrawIndexedIndirectCount-None-02692**
  If a **VkImageView** is sampled with **VK_FILTER_CUBIC_EXT** as a result of this command, then the image view's **format features** **must** contain **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT**.

• **VUID-vkCmdDrawIndexedIndirectCount-filterCubic-02694**
  Any **VkImageView** being sampled with **VK_FILTER_CUBIC_EXT** as a result of this command **must** have a **VkImageViewType** and format that supports cubic filtering, as specified by **VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic** returned by **vkGetPhysicalDeviceImageFormatProperties2**.

• **VUID-vkCmdDrawIndexedIndirectCount-filterCubicMinmax-02695**
  Any **VkImageView** being sampled with **VK_FILTER_CUBIC_EXT** with a reduction mode of either **VK_SAMPLER_REDUCTION_MODE_MIN** or **VK_SAMPLER_REDUCTION_MODE_MAX** as a result of this command **must** have a **VkImageViewType** and format that supports cubic filtering together with minmax filtering, as specified by **VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax** returned by **vkGetPhysicalDeviceImageFormatProperties2**.

• **VUID-vkCmdDrawIndexedIndirectCount-flags-02696**
  Any **VkImage** created with a **VkImageCreateInfo::flags** containing **VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV** sampled as a result of this command **must** only be sampled using a **VkSamplerAddressMode** of **VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE**.

• **VUID-vkCmdDrawIndexedIndirectCount-None-02697**
  For each set **n** that is statically used by the **VkPipeline** bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to **n** at the same pipeline bind point, with a **VkPipelineLayout** that is compatible for set **n**, with the **VkPipelineLayout** used to create the current **VkPipeline**, as described in **Pipeline Layout Compatibility**.

• **VUID-vkCmdDrawIndexedIndirectCount-None-02698**
  For each push constant that is statically used by the **VkPipeline** bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a **VkPipelineLayout** that is compatible for push constants, with the **VkPipelineLayout** used to create the current **VkPipeline**, as described in **Pipeline Layout Compatibility**.
Compatibility

- VUID-vkCmdDrawIndexedIndirectCount-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdDrawIndexedIndirectCount-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdDrawIndexedIndirectCount-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdDrawIndexedIndirectCount-None-02782
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdDrawIndexedIndirectCount-None-02783
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdDrawIndexedIndirectCount-None-02784
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdDrawIndexedIndirectCount-None-02785
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDrawIndexedIndirectCount-None-02786
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

- VUID-vkCmdDrawIndexedIndirectCount-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- VUID-vkCmdDrawIndexedIndirectCount-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- VUID-vkCmdDrawIndexedIndirectCount-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndexedIndirectCount-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndexedIndirectCount-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawIndexedIndirectCount-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawIndexedIndirectCount-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndexedIndirectCount-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawIndexedIndirectCount-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawIndexedIndirectCount-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawIndexedIndirectCount-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a...
• VUID-vkCmdDrawIndexedIndirectCount-None-04584

Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the **Render Pass** chapter.

• VUID-vkCmdDrawIndexedIndirectCount-maxMultiviewInstanceIndex-02688

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`

• VUID-vkCmdDrawIndexedIndirectCount-sampleLocationsEnable-02689

If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

• VUID-vkCmdDrawIndexedIndirectCount-viewportCount-03417

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

• VUID-vkCmdDrawIndexedIndirectCount-scissorCount-03418

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

• VUID-vkCmdDrawIndexedIndirectCount-viewportCount-03419

If the bound graphics pipeline state was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

• VUID-vkCmdDrawIndexedIndirectCount-viewportCount-04137

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled, then the bound graphics pipeline **must** have been created with `VkPipelineViewportWScalingStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

Chapter 21. Drawing Commands | 1359
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then...
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirectCount-None-04876
  
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT dynamic state enabled then vkCmdSetRasterizerDiscardEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirectCount-None-04877
  
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT dynamic state enabled then vkCmdSetDepthBiasEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirectCount-logicOp-04878
  
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LOGIC_OP_EXT dynamic state enabled then vkCmdSetLogicOpEXT must have been called in the current command buffer prior to this draw command and the logicOp must be a valid VkLogicOp value

- VUID-vkCmdDrawIndexedIndirectCount-None-04879
  
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT dynamic state enabled then vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndexedIndirectCount-primitiveFragmentShadingRateWithMultipleViewports-04552
  
  If the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the PrimitiveShadingRateKHR built-in, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must be 1

- VUID-vkCmdDrawIndexedIndirectCount-blendEnable-04727
  
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDrawIndexedIndirectCount-rasterizationSamples-04740
  
  If rasterization is not disabled in the bound graphics pipeline, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndexedIndirectCount-pStrides-04884
  
  If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of...
**vkCmdBindVertexBuffer2EXT** must not be **NULL**

- **VUID-vkCmdDrawIndexedIndirectCount-None-04007**
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or **VK_NULL_HANDLE** buffers bound

- **VUID-vkCmdDrawIndexedIndirectCount-None-04008**
  If the **nullDescriptor** feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be **VK_NULL_HANDLE**

- **VUID-vkCmdDrawIndexedIndirectCount-None-02721**
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in **Vertex Input Description**

- **VUID-vkCmdDrawIndexedIndirectCount-buffer-02708**
  If **buffer** is non-sparse then it must be bound completely and contiguous to a single **VkDeviceMemory** object

- **VUID-vkCmdDrawIndexedIndirectCount-buffer-02709**
  **buffer** must have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set

- **VUID-vkCmdDrawIndexedIndirectCount-offset-02710**
  **offset** must be a multiple of 4

- **VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-02711**
  **commandBuffer** must not be a protected command buffer

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02714**
  If **countBuffer** is non-sparse then it must be bound completely and contiguous to a single **VkDeviceMemory** object

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02715**
  **countBuffer** must have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set

- **VUID-vkCmdDrawIndexedIndirectCount-countBufferOffset-02716**
  **countBufferOffset** must be a multiple of 4

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-02717**
  The count stored in **countBuffer** must be less than or equal to **VkPhysicalDeviceLimits::maxDrawIndirectCount**

- **VUID-vkCmdDrawIndexedIndirectCount-countBufferOffset-04129**
  
  (**countBufferOffset** + sizeof(uint32_t)) must be less than or equal to the size of **countBuffer**

- **VUID-vkCmdDrawIndexedIndirectCount-stride-03142**
  **stride** must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndexedIndirectCommand)

- **VUID-vkCmdDrawIndexedIndirectCount-maxDrawCount-03143**
  
  If **maxDrawCount** is greater than or equal to 1, (**stride** × (**maxDrawCount** - 1) + **offset** + sizeof(VkDrawIndexedIndirectCommand)) must be less than or equal to the size of **buffer**

- **VUID-vkCmdDrawIndexedIndirectCount-countBuffer-03153**
  If count stored in **countBuffer** is equal to 1, (**offset** + sizeof(VkDrawIndexedIndirectCommand))
must be less than or equal to the size of buffer

• VUID-vkCmdDrawIndexedIndirectCount-countBuffer-03154
  If count stored in countBuffer is greater than 1, \((\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof}(	ext{VkDrawIndexedIndirectCommand}))\) must be less than or equal to the size of buffer

Valid Usage (Implicit)

• VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdDrawIndexedIndirectCount-buffer-parameter
  buffer must be a valid VkBuffer handle

• VUID-vkCmdDrawIndexedIndirectCount-countBuffer-parameter
  countBuffer must be a valid VkBuffer handle

• VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdDrawIndexedIndirectCount-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

• VUID-vkCmdDrawIndexedIndirectCount-renderpass
  This command must only be called inside of a render pass instance

• VUID-vkCmdDrawIndexedIndirectCount-commonparent
  Each of buffer, commandBuffer, and countBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21.3.1. Drawing Transform Feedback

It is possible to draw vertex data that was previously captured during active transform feedback by
binding one or more of the transform feedback buffers as vertex buffers. A pipeline barrier is required between using the buffers as transform feedback buffers and vertex buffers to ensure all writes to the transform feedback buffers are visible when the data is read as vertex attributes. The source access is `VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT` and the destination access is `VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT` for the pipeline stages `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT` and `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` respectively. The value written to the counter buffer by `vkCmdEndTransformFeedbackEXT` can be used to determine the vertex count for the draw. A pipeline barrier is required between using the counter buffer for `vkCmdEndTransformFeedbackEXT` and `vkCmdDrawIndirectByteCountEXT` where the source access is `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT` and the destination access is `VK_ACCESS_INDIRECT_COMMAND_READ_BIT` for the pipeline stages `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT` and `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT` respectively.

To record a non-indexed draw call, where the vertex count is based on a byte count read from a buffer and the passed in vertex stride parameter, call:

```c
// Provided by VK_EXT_transform_feedback
void vkCmdDrawIndirectByteCountEXT(
    VkCommandBuffer commandBuffer,           commandBuffer,
    uint32_t instanceCount,                  instanceCount,
    uint32_t firstInstance,                  firstInstance,
    VkBuffer counterBuffer,                  counterBuffer,
    VkDeviceSize counterBufferOffset,        counterBufferOffset,
    uint32_t counterOffset,                  counterOffset,
    uint32_t vertexStride);                   vertexStride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `instanceCount` is the number of instances to draw.
- `firstInstance` is the instance ID of the first instance to draw.
- `counterBuffer` is the buffer handle from where the byte count is read.
- `counterBufferOffset` is the offset into the buffer used to read the byte count, which is used to calculate the vertex count for this draw call.
- `counterOffset` is subtracted from the byte count read from the `counterBuffer` at the `counterBufferOffset`.
- `vertexStride` is the stride in bytes between each element of the vertex data that is used to calculate the vertex count from the counter value. This value is typically the same value that was used in the graphics pipeline state when the transform feedback was captured as the `XfbStride`.

When the command is executed, primitives are assembled in the same way as done with `vkCmdDraw` except the `vertexCount` is calculated based on the byte count read from `counterBuffer` at offset `counterBufferOffset`. The assembled primitives execute the bound graphics pipeline.

The effective `vertexCount` is calculated as follows:
The effective `firstVertex` is zero.
Valid Usage

- **VUID-vkCmdDrawIndirectByteCountEXT-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndirectByteCountEXT-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- **VUID-vkCmdDrawIndirectByteCountEXT-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDrawIndirectByteCountEXT-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDrawIndirectByteCountEXT-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.
Compatibility

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawIndirectByteCountEXT-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02702**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02703**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02704**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02705**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02706**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawIndirectByteCountEXT-commandBuffer-02707**
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
A `VkPipeline` object bound to the pipeline bind point used by this command **must** not be a protected resource.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-04115**
  If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction **must** have at least as many components as the image view's format.

- **VUID-vkCmdDrawIndirectByteCountEXT-OpImageWrite-04469**
  If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction **must** have at least as many components as the buffer view's format.

- **VUID-vkCmdDrawIndirectByteCountEXT-SampledType-04470**
  If a `VkImageView` with a `VkFormat` that has a 64-bit channel width is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction **must** have a `Width` of 64.

- **VUID-vkCmdDrawIndirectByteCountEXT-SampledType-04471**
  If a `VkImageView` with a `VkFormat` that has a channel width less than 64-bit is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction **must** have a `Width` of 32.

- **VUID-vkCmdDrawIndirectByteCountEXT-SampledType-04472**
  If a `VkBufferView` with a `VkFormat` that has a 64-bit channel width is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction **must** have a `Width` of 64.

- **VUID-vkCmdDrawIndirectByteCountEXT-SampledType-04473**
  If a `VkBufferView` with a `VkFormat` that has a channel width less than 64-bit is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction **must** have a `Width` of 32.

- **VUID-vkCmdDrawIndirectByteCountEXT-sparseImageInt64Atomics-04474**
  If the `sparseImageInt64Atomics` feature is not enabled, `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag **must** not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

- **VUID-vkCmdDrawIndirectByteCountEXT-sparseImageInt64Atomics-04475**
  If the `sparseImageInt64Atomics` feature is not enabled, `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag **must** not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

- **VUID-vkCmdDrawIndirectByteCountEXT-renderPass-02684**
  The current render pass **must be compatible** with the `renderPass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

- **VUID-vkCmdDrawIndirectByteCountEXT-subpass-02685**
  The subpass index of the current render pass **must** be equal to the `subpass` member of the `VkGraphicsPipelineCreateInfo` structure specified when creating the `VkPipeline` bound to `VK_PIPELINE_BIND_POINT_GRAPHICS`.

- **VUID-vkCmdDrawIndirectByteCountEXT-None-02686**
  Every input attachment used by the current subpass **must** be bound to the pipeline via a...
Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set.

If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

If the bound graphics pipeline was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the scissorCount parameter of vkCmdSetScissorWithCountEXT.

If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV dynamic state enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then...
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectByteCountEXT-None-04876
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT dynamic state enabled then vkCmdSetRasterizerDiscardEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectByteCountEXT-None-04877
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT dynamic state enabled then vkCmdSetDepthBiasEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectByteCountEXT-logicOp-04878
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LOGIC_OP_EXT dynamic state enabled then vkCmdSetLogicOpEXT must have been called in the current command buffer prior to this draw command and the logicOp must be a valid VkLogicOp value

- VUID-vkCmdDrawIndirectByteCountEXT-None-04879
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT dynamic state enabled then vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawIndirectByteCountEXT-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the PrimitiveShadingRateKHR built-in, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must be 1

- VUID-vkCmdDrawIndirectByteCountEXT-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDrawIndirectByteCountEXT-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawIndirectByteCountEXT-pStrides-04884
  If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of...
vkCmdBindVertexBuffers2EXT must not be NULL

- VUID-vkCmdDrawIndirectByteCountEXT-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must have either valid or VK_NULL_HANDLE buffers bound

- VUID-vkCmdDrawIndirectByteCountEXT-None-04008
  If the nullDescriptor feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point’s interface must not be VK_NULL_HANDLE

- VUID-vkCmdDrawIndirectByteCountEXT-None-02721
  For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Vertex Input Description

- VUID-vkCmdDrawIndirectByteCountEXT-transformFeedback-02287
  VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback must be enabled

- VUID-vkCmdDrawIndirectByteCountEXT-transformFeedbackDraw-02288
  The implementation must support VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackDraw

- VUID-vkCmdDrawIndirectByteCountEXT-vertexStride-02289
  vertexStride must be greater than 0 and less than or equal to VkPhysicalDeviceLimits::maxTransformFeedbackBufferDataStride

- VUID-vkCmdDrawIndirectByteCountEXT-counterBuffer-04567
  If counterBuffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdDrawIndirectByteCountEXT-counterBuffer-02290
  counterBuffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-vkCmdDrawIndirectByteCountEXT-counterBufferOffset-04568
  counterBufferOffset must be a multiple of 4

- VUID-vkCmdDrawIndirectByteCountEXT-commandBuffer-02646
  commandBuffer must not be a protected command buffer
Valid Usage (Implicit)

- **VUID-vkCmdDrawIndirectByteCountEXT-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdDrawIndirectByteCountEXT-counterBuffer-parameter**
  
  `counterBuffer` must be a valid `VkBuffer` handle

- **VUID-vkCmdDrawIndirectByteCountEXT-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdDrawIndirectByteCountEXT-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- **VUID-vkCmdDrawIndirectByteCountEXT-renderpass**
  
  This command must only be called inside of a render pass instance

- **VUID-vkCmdDrawIndirectByteCountEXT-commonparent**
  
  Both of `commandBuffer`, and `counterBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21.4. Conditional Rendering

Certain rendering commands can be executed conditionally based on a value in buffer memory. These rendering commands are limited to drawing commands, dispatching commands, and clearing attachments with `vkCmdClearAttachments` within a conditional rendering block which is defined by commands `vkCmdBeginConditionalRenderingEXT` and `vkCmdEndConditionalRenderingEXT`. Other rendering commands remain unaffected by conditional rendering.

After beginning conditional rendering, it is considered active within the command buffer it was called until it is ended with `vkCmdEndConditionalRenderingEXT`. 
Conditional rendering **must** begin and end in the same command buffer. When conditional rendering is active, a primary command buffer **can** execute secondary command buffers if the **inherited conditional rendering** feature is enabled. For a secondary command buffer to be executed while conditional rendering is active in the primary command buffer, it **must** set the `conditionalRenderingEnable` flag of `VkCommandBufferInheritanceConditionalRenderingInfoEXT`, as described in the **Command Buffer Recording** section.

Conditional rendering **must** also either begin and end inside the same subpass of a render pass instance, or **must** both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

To begin conditional rendering, call:

```c
// Provided by VK_EXT_conditional_rendering
void vkCmdBeginConditionalRenderingEXT(
    VkCommandBuffer                             commandBuffer,
    const VkConditionalRenderingBeginInfoEXT*   pConditionalRenderingBegin);
```

- `commandBuffer` is the command buffer into which this command will be recorded.
- `pConditionalRenderingBegin` is a pointer to a `VkConditionalRenderingBeginInfoEXT` structure specifying parameters of conditional rendering.

**Valid Usage**

- VUID-vkCmdBeginConditionalRenderingEXT-None-01980
  Conditional rendering **must** not already be **active**

**Valid Usage (Implicit)**

- VUID-vkCmdBeginConditionalRenderingEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- VUID-vkCmdBeginConditionalRenderingEXT-pConditionalRenderingBegin-parameter
  `pConditionalRenderingBegin` **must** be a valid pointer to a valid `VkConditionalRenderingBeginInfoEXT` structure
- VUID-vkCmdBeginConditionalRenderingEXT-commandBuffer-recording
  `commandBuffer` **must** be in the **recording** state
- VUID-vkCmdBeginConditionalRenderingEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td>Both</td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The `VkConditionalRenderingBeginInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_conditional_rendering
typedef struct VkConditionalRenderingBeginInfoEXT {
    VkStructureType                   sType;
    const void*                       pNext;
    VkBuffer                          buffer;
    VkDeviceSize                      offset;
    VkConditionalRenderingFlagsEXT    flags;
} VkConditionalRenderingBeginInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `buffer` is a buffer containing the predicate for conditional rendering.
- `offset` is the byte offset into `buffer` where the predicate is located.
- `flags` is a bitmask of `VkConditionalRenderingFlagsEXT` specifying the behavior of conditional rendering.

If the 32-bit value at `offset` in `buffer` memory is zero, then the rendering commands are discarded, otherwise they are executed as normal. If the value of the predicate in buffer memory changes while conditional rendering is active, the rendering commands may be discarded in an implementation-dependent way. Some implementations may latch the value of the predicate upon beginning conditional rendering while others may read it before every rendering command.
Valid Usage

- **VUID-VkConditionalRenderingBeginInfoEXT-buffer-01981**
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

- **VUID-VkConditionalRenderingBeginInfoEXT-buffer-01982**
  buffer must have been created with the VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT bit set.

- **VUID-VkConditionalRenderingBeginInfoEXT-offset-01983**
  offset must be less than the size of buffer by at least 32 bits.

- **VUID-VkConditionalRenderingBeginInfoEXT-offset-01984**
  offset must be a multiple of 4.

Valid Usage (Implicit)

- **VUID-VkConditionalRenderingBeginInfoEXT-sType-sType**
  sType must be VK_STRUCTURE_TYPE_CONDITIONAL_RENDERING_BEGIN_INFO_EXT.

- **VUID-VkConditionalRenderingBeginInfoEXT-pNext-pNext**
  pNext must be NULL.

- **VUID-VkConditionalRenderingBeginInfoEXT-buffer-parameter**
  buffer must be a valid VkBuffer handle.

- **VUID-VkConditionalRenderingBeginInfoEXT-flags-parameter**
  flags must be a valid combination of VkConditionalRenderingFlagBitsEXT values.

Bits which can be set in `vkCmdBeginConditionalRenderingEXT::flags` specifying the behavior of conditional rendering are:

```c
// Provided by VK_EXT_conditional_rendering
typedef enum VkConditionalRenderingFlagBitsEXT {
    VK_CONDITIONAL_RENDERING_INVERTED_BIT_EXT = 0x00000001,
} VkConditionalRenderingFlagBitsEXT;
```

- **VK_CONDITIONAL_RENDERING_INVERTED_BIT_EXT** specifies the condition used to determine whether to discard rendering commands or not. That is, if the 32-bit predicate read from buffer memory at offset is zero, the rendering commands are not discarded, and if non zero, then they are discarded.

```c
// Provided by VK_EXT_conditional_rendering
typedef VkFlags VkConditionalRenderingFlagsEXT;
```

`VkConditionalRenderingFlagsEXT` is a bitmask type for setting a mask of zero or more `VkConditionalRenderingFlagBitsEXT`.

1376 | Chapter 21. Drawing Commands
To end conditional rendering, call:

```c
// Provided by VK_EXT_conditional_rendering
void vkCmdEndConditionalRenderingEXT(
    VkCommandBuffer commandBuffer);
```

- `commandBuffer` is the command buffer into which this command will be recorded.

Once ended, conditional rendering becomes inactive.

### Valid Usage

- VUID-vkCmdEndConditionalRenderingEXT-None-01985
  Conditional rendering **must** be active

- VUID-vkCmdEndConditionalRenderingEXT-None-01986
  If conditional rendering was made active outside of a render pass instance, it **must** not be ended inside a render pass instance

- VUID-vkCmdEndConditionalRenderingEXT-None-01987
  If conditional rendering was made active within a subpass it **must** be ended in the same subpass

### Valid Usage (Implicit)

- VUID-vkCmdEndConditionalRenderingEXT-commandBuffer-parameter
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- VUID-vkCmdEndConditionalRenderingEXT-commandBuffer-recording
  `commandBuffer` **must** be in the recording state

- VUID-vkCmdEndConditionalRenderingEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized
### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>Graphics, Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 21.5. Programmable Mesh Shading

In this drawing approach, primitives are assembled by the mesh shader stage. Mesh shading operates similarly to dispatching compute as the shaders make use of workgroups.

To record a draw that uses the mesh pipeline, call:

```c
// Provided by VK_NV_mesh_shader
void vkCmdDrawMeshTasksNV(
    VkCommandBuffer commandBuffer,
    uint32_t taskCount,
    uint32_t firstTask);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **taskCount** is the number of local workgroups to dispatch in the X dimension. Y and Z dimension are implicitly set to one.
- **firstTask** is the X component of the first workgroup ID.

When the command is executed, a global workgroup consisting of **taskCount** local workgroups is assembled.
Valid Usage

- **VUID-vkCmdDrawMeshTasksNV-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawMeshTasksNV-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdDrawMeshTasksNV-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

- **VUID-vkCmdDrawMeshTasksNV-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`.

- **VUID-vkCmdDrawMeshTasksNV-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDrawMeshTasksNV-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdDrawMeshTasksNV-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

- **VUID-vkCmdDrawMeshTasksNV-None-02697**
  For each set $n$ that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to $n$ at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set $n$, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in `Pipeline Layout Compatibility`.

- **VUID-vkCmdDrawMeshTasksNV-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in `Pipeline Layout`.
Compatibility

- **VUID-vkCmdDrawMeshTasksNV-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawMeshTasksNV-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawMeshTasksNV-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawMeshTasksNV-None-02702**
  If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawMeshTasksNV-None-02703**
  If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawMeshTasksNV-None-02859**
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDrawMeshTasksNV-None-02704**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDrawMeshTasksNV-None-02705**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDrawMeshTasksNV-None-02706**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- **VUID-vkCmdDrawMeshTasksNV-None-02707**
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
 VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

- **VUID-vkCmdDrawMeshTasksNV-None-04115**
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- **VUID-vkCmdDrawMeshTasksNV-OpImageWrite-04469**
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- **VUID-vkCmdDrawMeshTasksNV-SampledType-04470**
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- **VUID-vkCmdDrawMeshTasksNV-SampledType-04471**
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- **VUID-vkCmdDrawMeshTasksNV-SampledType-04472**
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- **VUID-vkCmdDrawMeshTasksNV-SampledType-04473**
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- **VUID-vkCmdDrawMeshTasksNV-sparseImageInt64Atomics-04474**
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- **VUID-vkCmdDrawMeshTasksNV-sparseImageInt64Atomics-04475**
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- **VUID-vkCmdDrawMeshTasksNV-renderPass-02684**
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- **VUID-vkCmdDrawMeshTasksNV-subpass-02685**
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- **VUID-vkCmdDrawMeshTasksNV-None-02686**
  Every input attachment used by the current subpass must be bound to the pipeline via a
Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set.

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of VkPipelineViewportStateCreateInfo::viewportCount and the scissorCount parameter of VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of VkPipelineViewportStateCreateInfo::viewportCount and the scissorCount parameter of VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV dynamic state enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then...
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksNV-None-04876
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT dynamic state enabled then vkCmdSetRasterizerDiscardEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksNV-None-04877
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT dynamic state enabled then vkCmdSetDepthBiasEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksNV-logicOp-04878
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LOGIC_OP_EXT dynamic state enabled then vkCmdSetLogicOpEXT must have been called in the current command buffer prior to this draw command and the logicOp must be a valid VkLogicOp value

- VUID-vkCmdDrawMeshTasksNV-None-04879
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT dynamic state enabled then vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksNV-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the PrimitiveShadingRateKHR built-in, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must be 1

- VUID-vkCmdDrawMeshTasksNV-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDrawMeshTasksNV-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawMeshTasksNV-pStrides-04884
  If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of
**vkCmdBindVertexBuffers2EXT** must not be **NULL**

- VUID-vkCmdDrawMeshTasksNV-taskCount-02119
  taskCount must be less than or equal to **VkPhysicalDeviceMeshShaderPropertiesNV::maxDrawMeshTasksCount**

### Valid Usage (Implicit)

- VUID-vkCmdDrawMeshTasksNV-commandBuffer-parameter
  commandBuffer must be a valid **VkCommandBuffer** handle

- VUID-vkCmdDrawMeshTasksNV-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDrawMeshTasksNV-commandBuffer-cmdpool
  The **VkCommandPool** that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDrawMeshTasksNV-renderpass
  This command must only be called inside of a render pass instance

### Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the **VkCommandPool** that commandBuffer was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indirect mesh tasks draw, call:

```c
// Provided by VK_NV_mesh_shader
void vkCmdDrawMeshTasksIndirectNV(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset,
    uint32_t                                    drawCount,
    uint32_t                                    stride);
```

- **commandBuffer** is the command buffer into which the command is recorded.
• **buffer** is the buffer containing draw parameters.
• **offset** is the byte offset into buffer where parameters begin.
• **drawCount** is the number of draws to execute, and **can** be zero.
• **stride** is the byte stride between successive sets of draw parameters.

`vkCmdDrawMeshTasksIndirectNV` behaves similarly to `vkCmdDrawMeshTasksNV` except that the parameters are read by the device from a buffer during execution. `drawCount` draws are executed by the command, with parameters taken from `buffer` starting at `offset` and increasing by `stride` bytes for each successive draw. The parameters of each draw are encoded in an array of `VkDrawMeshTasksIndirectCommandNV` structures. If `drawCount` is less than or equal to one, `stride` is ignored.
Valid Usage

- **VUID-vkCmdDrawMeshTasksIndirectNV-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawMeshTasksIndirectNV-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- **VUID-vkCmdDrawMeshTasksIndirectNV-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDrawMeshTasksIndirectNV-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDrawMeshTasksIndirectNV-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout
Compatibility

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawMeshTasksIndirectNV-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02859**
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02702**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02703**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02704**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02705**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawMeshTasksIndirectNV-None-02706**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawMeshTasksIndirectNV-commandBuffer-02707**
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
A VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

- VUID-vkCmdDrawMeshTasksIndirectNV-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- VUID-vkCmdDrawMeshTasksIndirectNV-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- VUID-vkCmdDrawMeshTasksIndirectNV-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawMeshTasksIndirectNV-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawMeshTasksIndirectNV-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawMeshTasksIndirectNV-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawMeshTasksIndirectNV-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawMeshTasksIndirectNV-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawMeshTasksIndirectNV-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawMeshTasksIndirectNV-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS.

- VUID-vkCmdDrawMeshTasksIndirectNV-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a...
Image subresources used as attachments in the current render pass **must** not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index **must** be less than or equal to `VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex`

If the bound graphics pipeline was created with `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` set to `VK_TRUE` and the current subpass has a depth/stencil attachment, then that attachment **must** have been created with the `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT` bit set.

If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetViewportWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, then `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT` **must** match the `VkPipelineViewportStateCreateInfo::viewportCount` of the pipeline.

If the bound graphics pipeline was created with both the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic states enabled then both `vkCmdSetViewportWithCountEXT` and `vkCmdSetScissorWithCountEXT` **must** have been called in the current command buffer prior to this draw command, and the `viewportCount` parameter of `vkCmdSetViewportWithCountEXT` **must** match the `scissorCount` parameter of `vkCmdSetScissorWithCountEXT`.

If the bound graphics pipeline was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled, then the bound graphics pipeline **must** have been created with `VkPipelineViewportWScalingStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportWithSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportWithExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then
vkCmdSetPatchControlPointsEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksIndirectNV-None-04876

  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT dynamic state enabled then vkCmdSetRasterizerDiscardEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksIndirectNV-None-04877

  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT dynamic state enabled then vkCmdSetDepthBiasEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksIndirectNV-logicOp-04878

  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_LOGIC_OP_EXT dynamic state enabled then vkCmdSetLogicOpEXT must have been called in the current command buffer prior to this draw command and the logicOp must be a valid VkLogicOp value

- VUID-vkCmdDrawMeshTasksIndirectNV-PrimitiveRestartEnable-04879

  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT dynamic state enabled then vkCmdSetPrimitiveRestartEnableEXT must have been called in the current command buffer prior to this draw command

- VUID-vkCmdDrawMeshTasksIndirectNV-primitiveFragmentShadingRateWithMultipleViewports-04552

  If the primitiveFragmentShadingRateWithMultipleViewports limit is not supported, the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the PrimitiveShadingRateKHR built-in, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must be 1

- VUID-vkCmdDrawMeshTasksIndirectNV-blendEnable-04727

  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view's format features do not contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, then the blendEnable member of the corresponding element of the pAttachments member of pColorBlendState must be VK_FALSE

- VUID-vkCmdDrawMeshTasksIndirectNV-rasterizationSamples-04740

  If rasterization is not disabled in the bound graphics pipeline, and neither the VK_AMD_mixed_attachment_samples nor the VK_NV_framebuffer_mixed_samples extensions are enabled, then VkPipelineMultisampleStateCreateInfo::rasterizationSamples must be the same as the current subpass color and/or depth/stencil attachments

- VUID-vkCmdDrawMeshTasksIndirectNV-pStrides-04884

  If the bound graphics pipeline was created with the VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT dynamic state enabled, then vkCmdBindVertexBuffers2EXT must have been called in the current command buffer prior to this draw command, and the pStrides parameter of
vkCmdBindVertexBuffers2EXT must not be NULL

- VUID-vkCmdDrawMeshTasksIndirectNV-buffer-02708
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdDrawMeshTasksIndirectNV-buffer-02709
  buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-vkCmdDrawMeshTasksIndirectNV-offset-02710
  offset must be a multiple of 4

- VUID-vkCmdDrawMeshTasksIndirectNV-commandBuffer-02711
  commandBuffer must not be a protected command buffer

- VUID-vkCmdDrawMeshTasksIndirectNV-drawCount-02718
  If the multi-draw indirect feature is not enabled, drawCount must be 0 or 1

- VUID-vkCmdDrawMeshTasksIndirectNV-drawCount-02719
  drawCount must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount

- VUID-vkCmdDrawMeshTasksIndirectNV-drawCount-02146
  If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawMeshTasksIndirectCommandNV)

- VUID-vkCmdDrawMeshTasksIndirectNV-drawCount-02156
  If drawCount is equal to 1, (offset + sizeof(VkDrawMeshTasksIndirectCommandNV)) must be less than or equal to the size of buffer

- VUID-vkCmdDrawMeshTasksIndirectNV-drawCount-02157
  If drawCount is greater than 1, (stride \times (drawCount - 1) + offset + sizeof (VkDrawMeshTasksIndirectCommandNV)) must be less than or equal to the size of buffer

Valid Usage (Implicit)

- VUID-vkCmdDrawMeshTasksIndirectNV-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDrawMeshTasksIndirectNV-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-vkCmdDrawMeshTasksIndirectNV-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDrawMeshTasksIndirectNV-commandBuffer-cmpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDrawMeshTasksIndirectNV-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdDrawMeshTasksIndirectNV-commonparent
  Both of buffer, and commandBuffer must have been created, allocated, or retrieved from the same VkDevice
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDrawMeshTasksIndirectCommandNV` structure is defined as:

```c
typedef struct VkDrawMeshTasksIndirectCommandNV {
    uint32_t    taskCount;
    uint32_t    firstTask;
} VkDrawMeshTasksIndirectCommandNV;
```

- `taskCount` is the number of local workgroups to dispatch in the X dimension. Y and Z dimension are implicitly set to one.
- `firstTask` is the X component of the first workgroup ID.

The members of `VkDrawMeshTasksIndirectCommandNV` have the same meaning as the similarly named parameters of `vkCmdDrawMeshTasksNV`.

Valid Usage

- VUID-VkDrawMeshTasksIndirectCommandNV-taskCount-02175
  taskCount must be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxDrawMeshTasksCount`

To record an indirect mesh tasks draw with the draw count sourced from a buffer, call:
// Provided by VK_NV_mesh_shader

```c
void vkCmdDrawMeshTasksIndirectCountNV(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset,
    VkBuffer                                    countBuffer,
    VkDeviceSize                                countBufferOffset,
    uint32_t                                    maxDrawCount,
    uint32_t                                    stride);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `buffer` is the buffer containing draw parameters.
- `offset` is the byte offset into `buffer` where parameters begin.
- `countBuffer` is the buffer containing the draw count.
- `countBufferOffset` is the byte offset into `countBuffer` where the draw count begins.
- `maxDrawCount` specifies the maximum number of draws that will be executed. The actual number of executed draw calls is the minimum of the count specified in `countBuffer` and `maxDrawCount`.
- `stride` is the byte stride between successive sets of draw parameters.

`vkCmdDrawMeshTasksIndirectCountNV` behaves similarly to `vkCmdDrawMeshTasksIndirectNV` except that the draw count is read by the device from a buffer during execution. The command will read an unsigned 32-bit integer from `countBuffer` located at `countBufferOffset` and use this as the draw count.
Valid Usage

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
Compatibility

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02700**
  A valid pipeline **must** be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02859**
  There **must** not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02702**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02703**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02704**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02705**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02706**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-commandBuffer-02707**
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the...
VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- VUID-vkCmdDrawMeshTasksIndirectCountNV-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view's format.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view's format.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDrawMeshTasksIndirectCountNV-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-vkCmdDrawMeshTasksIndirectCountNV-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

- VUID-vkCmdDrawMeshTasksIndirectCountNV-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a
Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter.

If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex.

If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set.

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline.

If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the viewportCount parameter of vkCmdSetScissorWithCountEXT.

If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the viewportCount parameter of vkCmdSetScissorWithCountEXT.

If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT.
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then
\texttt{vkCmdSetPatchControlPointsEXT} must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-04876**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT} dynamic state enabled then \texttt{vkCmdSetRasterizerDiscardEnableEXT} must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-04877**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT} dynamic state enabled then \texttt{vkCmdSetDepthBiasEnableEXT} must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-logicOp-04878**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_LOGIC_OP_EXT} dynamic state enabled then \texttt{vkCmdSetLogicOpEXT} must have been called in the current command buffer prior to this draw command and the \texttt{logicOp} must be a valid \texttt{VkLogicOp} value

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-None-04879**
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT} dynamic state enabled then \texttt{vkCmdSetPrimitiveRestartEnableEXT} must have been called in the current command buffer prior to this draw command

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-primitiveFragmentShadingRateWithMultipleViewports-04552**
  If the \texttt{primitiveFragmentShadingRateWithMultipleViewports} limit is not supported, the bound graphics pipeline was created with the \texttt{VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT} dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the \texttt{PrimitiveShadingRateKHR} built-in, then \texttt{vkCmdSetViewportWithCountEXT} must have been called in the current command buffer prior to this draw command, and the \texttt{viewportCount} parameter of \texttt{vkCmdSetViewportWithCountEXT} must be 1

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-blendEnable-04727**
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s \texttt{format features} do not contain \texttt{VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT}, then the \texttt{blendEnable} member of the corresponding element of the \texttt{pAttachments} member of \texttt{pColorBlendState} must be \texttt{VK_FALSE}

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-rasterizationSamples-04740**
  If rasterization is not disabled in the bound graphics pipeline, and neither the \texttt{VK_AMD_mixed_attachment_samples} nor the \texttt{VK_NV_framebuffer_mixed_samples} extensions are enabled, then \texttt{VkPipelineMultisampleStateCreateInfo::rasterizationSamples} must be the same as the current subpass color and/or depth/stencil attachments

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-pStrides-04884**
  If the bound graphics pipeline was created with the \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT} dynamic state enabled, then \texttt{vkCmdBindVertexBuffers2EXT} must have been called in the current command buffer prior to this draw command, and the \texttt{pStrides} parameter of
vkCmdBindVertexBuffers2EXT **must** not be NULL

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-buffer-02708**
  If *buffer* is non-sparse then it **must** be bound completely and contiguously to a single *VkDeviceMemory* object

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-buffer-02709**
  *buffer* **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-offset-02710**
  *offset* **must** be a multiple of 4

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-commandBuffer-02711**
  *commandBuffer* **must** not be a protected command buffer

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBuffer-02714**
  If *countBuffer* is non-sparse then it **must** be bound completely and contiguously to a single *VkDeviceMemory* object

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBuffer-02715**
  *countBuffer* **must** have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBufferOffset-02716**
  *countBufferOffset* **must** be a multiple of 4

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBuffer-02717**
  The count stored in *countBuffer* **must** be less than or equal to `VkPhysicalDeviceLimits::maxDrawIndirectCount`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBufferOffset-04129**
  (*countBufferOffset + sizeof(uint32_t)) **must** be less than or equal to the size of *countBuffer*

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-stride-02182**
  *stride* **must** be a multiple of 4 and **must** be greater than or equal to `sizeof(VkDrawMeshTasksIndirectCommandNV)`

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-maxDrawCount-02183**
  If *maxDrawCount* is greater than or equal to 1, (*stride × (maxDrawCount - 1) + offset + sizeof(VkDrawMeshTasksIndirectCommandNV)) **must** be less than or equal to the size of *buffer*

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBuffer-02191**
  If the count stored in *countBuffer* is equal to 1, (*offset + sizeof(VkDrawMeshTasksIndirectCommandNV)) **must** be less than or equal to the size of *buffer*

- **VUID-vkCmdDrawMeshTasksIndirectCountNV-countBuffer-02192**
  If the count stored in *countBuffer* is greater than 1, (*stride × (drawCount - 1) + offset + sizeof(VkDrawMeshTasksIndirectCommandNV)) **must** be less than or equal to the size of *buffer*
Valid Usage (Implicit)

- VUID-vkCmdDrawMeshTasksIndirectCountNV-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDrawMeshTasksIndirectCountNV-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-vkCmdDrawMeshTasksIndirectCountNV-countBuffer-parameter
  countBuffer must be a valid VkBuffer handle

- VUID-vkCmdDrawMeshTasksIndirectCountNV-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDrawMeshTasksIndirectCountNV-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdDrawMeshTasksIndirectCountNV-renderpass
  This command must only be called inside of a render pass instance

- VUID-vkCmdDrawMeshTasksIndirectCountNV-commonparent
  Each of buffer, commandBuffer, and countBuffer must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 22. Fixed-Function Vertex Processing

Vertex fetching is controlled via configurable state, as a logically distinct graphics pipeline stage.

22.1. Vertex Attributes

Vertex shaders can define input variables, which receive vertex attribute data transferred from one or more VkBuffer(s) by drawing commands. Vertex shader input variables are bound to buffers via an indirect binding where the vertex shader associates a vertex input attribute number with each variable, vertex input attributes are associated to vertex input bindings on a per-pipeline basis, and vertex input bindings are associated with specific buffers on a per-draw basis via the vkCmdBindVertexBuffers command. Vertex input attribute and vertex input binding descriptions also contain format information controlling how data is extracted from buffer memory and converted to the format expected by the vertex shader.

There are VkPhysicalDeviceLimits::maxVertexInputAttributes number of vertex input attributes and VkPhysicalDeviceLimits::maxVertexInputBindings number of vertex input bindings (each referred to by zero-based indices), where there are at least as many vertex input attributes as there are vertex input bindings. Applications can store multiple vertex input attributes interleaved in a single buffer, and use a single vertex input binding to access those attributes.

In GLSL, vertex shaders associate input variables with a vertex input attribute number using the location layout qualifier. The component layout qualifier associates components of a vertex shader input variable with components of a vertex input attribute.

**GLSL example**

```glsl
// Assign location M to variableName
layout (location=M, component=2) in vec2 variableName;

// Assign locations [N,N+L) to the array elements of variableNameArray
layout (location=N) in vec4 variableNameArray[L];
```

In SPIR-V, vertex shaders associate input variables with a vertex input attribute number using the Location decoration. The Component decoration associates components of a vertex shader input variable with components of a vertex input attribute. The Location and Component decorations are specified via the OpDecorate instruction.
SPIR-V example

```
...%1 = OpExtInstImport "GLSL.std.450"
...
OpName %9 "variableName"
OpName %15 "variableNameArray"
OpDecorate %18 BuiltIn VertexIndex
OpDecorate %19 BuiltIn InstanceIndex
OpDecorate %9 Location M
OpDecorate %9 Component 2
OpDecorate %15 Location N
...
%2 = OpTypeVoid
%3 = OpTypeFunction %2
%6 = OpTypeFloat 32
%7 = OpTypeVector %6 2
%8 = OpTypePointer Input %7
%9 = OpVariable %8 Input
%10 = OpTypeVector %6 4
%11 = OpTypeInt 32 0
%12 = OpConstant %11 L
%13 = OpTypeArray %10 %12
%14 = OpTypePointer Input %13
%15 = OpVariable %14 Input
...
```

22.1.1. Attribute Location and Component Assignment

Vertex shaders allow **Location** and **Component** decorations on input variable declarations. The **Location** decoration specifies which vertex input attribute is used to read and interpret the data that a variable will consume. The **Component** decoration allows the location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable starting at component N will consume components N, N+1, N+2, ... up through its size. For single precision types, it is invalid if the sequence of components gets larger than 3.

When a vertex shader input variable declared using a scalar or vector 32-bit data type is assigned a location, its value(s) are taken from the components of the input attribute specified with the corresponding **VkVertexInputAttributeDescription::location**. The components used depend on the type of variable and the **Component** decoration specified in the variable declaration, as identified in Input attribute components accessed by 32-bit input variables. Any 32-bit scalar or vector input will consume a single location. For 32-bit data types, missing components are filled in with default values as described below.

*Table 31. Input attribute components accessed by 32-bit input variables*
32-bit data type | Component decoration | Components consumed
---|---|---
scalar | 0 or unspecified | (x, o, o, o)
scalar | 1 | (o, y, o, o)
scalar | 2 | (o, o, z, o)
scalar | 3 | (o, o, o, w)
two-component vector | 0 or unspecified | (x, y, o, o)
two-component vector | 1 | (o, y, z, o)
two-component vector | 2 | (o, o, z, w)
three-component vector | 0 or unspecified | (x, y, z, o)
three-component vector | 1 | (o, y, z, w)
four-component vector | 0 or unspecified | (x, y, z, w)

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input format (if present), or the default value.

When a vertex shader input variable declared using a 32-bit floating point matrix type is assigned a location \( i \), its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. Such matrices are treated as an array of column vectors with values taken from the input attributes identified in `Input attributes accessed by 32-bit input matrix variables`. The `VkVertexInputAttributeDescription::format` must be specified with a `VkFormat` that corresponds to the appropriate type of column vector. The Component decoration must not be used with matrix types.

### Table 32. Input attributes accessed by 32-bit input matrix variables

<table>
<thead>
<tr>
<th>Data type</th>
<th>Column vector type</th>
<th>Locations consumed</th>
<th>Components consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>mat2</td>
<td>two-component vector</td>
<td>( i, i+1 )</td>
<td>(x, y, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td>mat2x3</td>
<td>three-component vector</td>
<td>( i, i+1 )</td>
<td>(x, y, z, o), (x, y, z, o)</td>
</tr>
<tr>
<td>mat2x4</td>
<td>four-component vector</td>
<td>( i, i+1 )</td>
<td>(x, y, z, w), (x, y, z, w)</td>
</tr>
<tr>
<td>mat3x2</td>
<td>two-component vector</td>
<td>( i, i+1, i+2 )</td>
<td>(x, y, o, o), (x, y, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td>mat3</td>
<td>three-component vector</td>
<td>( i, i+1, i+2 )</td>
<td>(x, y, z, o), (x, y, z, o), (x, y, z, o)</td>
</tr>
<tr>
<td>mat3x4</td>
<td>four-component vector</td>
<td>( i, i+1, i+2 )</td>
<td>(x, y, z, w), (x, y, z, w), (x, y, z, w)</td>
</tr>
<tr>
<td>mat4x2</td>
<td>two-component vector</td>
<td>( i, i+1, i+2, i+3 )</td>
<td>(x, y, o, o), (x, y, o, o), (x, y, o, o), (x, y, o, o)</td>
</tr>
</tbody>
</table>
Components indicated by “o” are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input (if present), or the default value.

When a vertex shader input variable declared using a scalar or vector 64-bit data type is assigned a location \( i \), its values are taken from consecutive input attributes starting with the corresponding \texttt{VkVertexInputAttributeDescription::location}. The locations and components used depend on the type of variable and the Component decoration specified in the variable declaration, as identified in Table 33. Input attribute locations and components accessed by 64-bit input variables. For 64-bit data types, no default attribute values are provided. Input variables must not use more components than provided by the attribute. Input attributes which have one- or two-component 64-bit formats will consume a single location. Input attributes which have three- or four-component 64-bit formats will consume two consecutive locations. A 64-bit scalar data type will consume two components, and a 64-bit two-component vector data type will consume all four components available within a location. A three- or four-component 64-bit data type must not specify a component. A three-component 64-bit data type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations. A four-component 64-bit data type will consume all four components of the first location and all four components of the second location. It is invalid for a scalar or two-component 64-bit data type to specify a component of 1 or 3.

### Table 33. Input attribute locations and components accessed by 64-bit input variables

<table>
<thead>
<tr>
<th>Input format</th>
<th>Locations consumed</th>
<th>64-bit data type</th>
<th>Location decoration</th>
<th>Component decoration</th>
<th>32-bit component consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>R64</td>
<td>i</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, -, -)</td>
</tr>
<tr>
<td>R64G64</td>
<td>i</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w)</td>
</tr>
<tr>
<td>Input format</td>
<td>Locations consumed</td>
<td>64-bit data type</td>
<td>Location decoration</td>
<td>Component decoration</td>
<td>32-bit component consumed</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------</td>
<td>------------------</td>
<td>---------------------</td>
<td>----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>R64G64B64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w), (o, o, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>three-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, -, -)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R64G64B64A64</td>
<td>i, i+1</td>
<td>scalar</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, o, o), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i+1</td>
<td>2</td>
<td>(o, o, o, o), (o, o, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w), (o, o, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>two-component vector</td>
<td>i+1</td>
<td>0 or unspecified</td>
<td>(o, o, o, o), (x, y, z, w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>three-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, o, o)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>four-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, z, w)</td>
</tr>
</tbody>
</table>

Components indicated by “o” are available for use by other input variables which are sourced from the same attribute. Components indicated by “-” are not available for input variables as there are no default values provided for 64-bit data types, and there is no data provided by the input format.

When a vertex shader input variable declared using a 64-bit floating-point matrix type is assigned a location \(i\), its values are taken from consecutive input attribute locations. Such matrices are treated as an array of column vectors with values taken from the input attributes as shown in **Input attribute locations and components accessed by 64-bit input variables**. Each column vector starts at the location immediately following the last location of the previous column vector. The number of attributes and components assigned to each matrix is determined by the matrix dimensions and ranges from two to eight locations.
When a vertex shader input variable declared using an array type is assigned a location, its values are taken from consecutive input attributes starting with the corresponding `VkVertexInputAttributeDescription::location`. The number of attributes and components assigned to each element are determined according to the data type of the array elements and `Component` decoration (if any) specified in the declaration of the array, as described above. Each element of the array, in order, is assigned to consecutive locations, but all at the same specified component within each location.

Only input variables declared with the data types and component decorations as specified above are supported. Location aliasing is causing two variables to have the same location number. Component aliasing is assigning the same (or overlapping) component number for two location aliases. Location aliasing is allowed only if it does not cause component aliasing. Further, when location aliasing, the aliases sharing the location must all have the same SPIR-V floating-point component type or all have the same width integer-type components.

### 22.2. Vertex Input Description

Applications specify vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation. `VkGraphicsPipelineCreateInfo::pVertexInputState` is a pointer to a `VkPipelineVertexInputStateCreateInfo` value.

The `VkPipelineVertexInputStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkPipelineVertexInputStateCreateFlags       flags;
    uint32_t                                    vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription*      pVertexBindingDescriptions;
    uint32_t                                    vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription*    pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `vertexBindingDescriptionCount` is the number of vertex binding descriptions provided in `pVertexBindingDescriptions`.
- `pVertexBindingDescriptions` is a pointer to an array of `VkVertexInputBindingDescription` structures.
- `vertexAttributeDescriptionCount` is the number of vertex attribute descriptions provided in `pVertexAttributeDescriptions`.
- `pVertexAttributeDescriptions` is a pointer to an array of `VkVertexInputAttributeDescription` structures.
Valid Usage

- VUID-VkPipelineVertexInputStateCreateInfo-vertexBindingDescriptionCount-00613
  vertexBindingDescriptionCount must be less than or equal toVkPhysicalDeviceLimits::maxVertexInputBindings

- VUID-VkPipelineVertexInputStateCreateInfo-vertexAttributeDescriptionCount-00614
  vertexAttributeDescriptionCount must be less than or equal toVkPhysicalDeviceLimits::maxVertexInputAttributes

- VUID-VkPipelineVertexInputStateCreateInfo-binding-00615
  For every binding specified by each element of pVertexAttributeDescriptions, a
  VkVertexInputBindingDescription must exist in pVertexBindingDescriptions with the same
  value of binding

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexBindingDescriptions-00616
  All elements of pVertexBindingDescriptions must describe distinct binding numbers

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexAttributeDescriptions-00617
  All elements of pVertexAttributeDescriptions must describe distinct attribute locations

Valid Usage (Implicit)

- VUID-VkPipelineVertexInputStateCreateInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO

- VUID-VkPipelineVertexInputStateCreateInfo-pNext-pNext
  pNext must be NULL or a pointer to a valid instance of
  VkPipelineVertexInputDivisorStateCreateInfoEXT

- VUID-VkPipelineVertexInputStateCreateInfo-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkPipelineVertexInputStateCreateInfo-inputCreateInfo-flags-zero
  flags must be 0

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexBindingDescriptions-parameter
  If vertexBindingDescriptionCount is not 0, pVertexBindingDescriptions must be a valid
  pointer to an array of vertexBindingDescriptionCount valid
  VkVertexInputBindingDescription structures

- VUID-VkPipelineVertexInputStateCreateInfo-pVertexAttributeDescriptions-parameter
  If vertexAttributeDescriptionCount is not 0, pVertexAttributeDescriptions must be a valid
  pointer to an array of vertexAttributeDescriptionCount valid
  VkVertexInputAttributeDescription structures

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineVertexInputStateCreateFlags;

VkPipelineVertexInputStateCreateFlags is a bitmask type for setting a mask, but is currently
Each vertex input binding is specified by the `VkVertexInputBindingDescription` structure, defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkVertexInputBindingDescription {
    uint32_t binding;
    uint32_t stride;
    VkVertexInputRate inputRate;
} VkVertexInputBindingDescription;
```

- `binding` is the binding number that this structure describes.
- `stride` is the distance in bytes between two consecutive elements within the buffer.
- `inputRate` is a `VkVertexInputRate` value specifying whether vertex attribute addressing is a function of the vertex index or of the instance index.

### Valid Usage

- VUID-VkVertexInputBindingDescription-binding-00618
  - `binding` must be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- VUID-VkVertexInputBindingDescription-stride-00619
  - `stride` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindingStride`
- VUID-VkVertexInputBindingDescription-stride-04456
  - If the `VK_KHR_portability_subset` extension is enabled, `stride` must be a multiple of, and at least as large as, `VkPhysicalDevicePortabilitySubsetPropertiesKHR::minVertexInputBindingStrideAlignment`.

### Valid Usage (Implicit)

- VUID-VkVertexInputBindingDescription-inputRate-parameter
  - `inputRate` must be a valid `VkVertexInputRate` value

Possible values of `VkVertexInputBindingDescription::inputRate`, specifying the rate at which vertex attributes are pulled from buffers, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkVertexInputRate {
    VK_VERTEX_INPUT_RATE_VERTEX = 0,
    VK_VERTEX_INPUT_RATE_INSTANCE = 1,
} VkVertexInputRate;
```

- `VK_VERTEX_INPUT_RATE_VERTEX` specifies that vertex attribute addressing is a function of the vertex index.
• **VK_VERTEX_INPUT_RATE_INSTANCE** specifies that vertex attribute addressing is a function of the instance index.

Each vertex input attribute is specified by the `VkVertexInputAttributeDescription` structure, defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkVertexInputAttributeDescription {
    uint32_t location;
    uint32_t binding;
    VkFormat format;
    uint32_t offset;
} VkVertexInputAttributeDescription;
```

- **location** is the shader input location number for this attribute.
- **binding** is the binding number which this attribute takes its data from.
- **format** is the size and type of the vertex attribute data.
- **offset** is a byte offset of this attribute relative to the start of an element in the vertex input binding.

### Valid Usage

- VUID-VkVertexInputAttributeDescription-location-00620
  
  **location must be less than** `VkPhysicalDeviceLimits::maxVertexInputAttributes`

- VUID-VkVertexInputAttributeDescription-binding-00621
  
  **binding must be less than** `VkPhysicalDeviceLimits::maxVertexInputBindings`

- VUID-VkVertexInputAttributeDescription-offset-00622
  
  **offset must be less than or equal to** `VkPhysicalDeviceLimits::maxVertexInputAttributeOffset`

- VUID-VkVertexInputAttributeDescription-format-00623
  
  **format must be allowed as a vertex buffer format, as specified by the** `VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT` **flag in** `VkFormatProperties::bufferFeatures` **returned by** `vkGetPhysicalDeviceFormatProperties`

- VUID-VkVertexInputAttributeDescription-vertexAttributeAccessBeyondStride-04457
  
  If the **VK_KHR_portability_subset** extension is enabled, and **VkPhysicalDevicePortabilitySubsetFeaturesKHR::vertexAttributeAccessBeyondStride is VK_FALSE**, the sum of **offset** plus the size of the vertex attribute data described by **format** **must not be greater than** **stride** in the **VkVertexInputBindingDescription** referenced in **binding**.
An alternative to specifying the vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation, the pipeline can be created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_EXT` dynamic state enabled, and for that state to be set dynamically with:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
void vkCmdSetVertexInputEXT(
    VkCommandBuffer commandBuffer,
    uint32_t vertexBindingDescriptionCount,
    const VkVertexInputBindingDescription2EXT* pVertexBindingDescriptions,
    uint32_t vertexAttributeDescriptionCount,
    const VkVertexInputAttributeDescription2EXT* pVertexAttributeDescriptions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `vertexBindingDescriptionCount` is the number of vertex binding descriptions provided in `pVertexBindingDescriptions`.
- `pVertexBindingDescriptions` is a pointer to an array of `VkVertexInputBindingDescription2EXT` structures.
- `vertexAttributeDescriptionCount` is the number of vertex attribute descriptions provided in `pVertexAttributeDescriptions`.
- `pVertexAttributeDescriptions` is a pointer to an array of `VkVertexInputAttributeDescription2EXT` structures.

This command sets the vertex input attribute and vertex input binding descriptions state for subsequent draw commands.
Valid Usage

- **VUID-vkCmdSetVertexInputEXT-None-04790**
  The `vertexInputDynamicState` feature must be enabled

- **VUID-vkCmdSetVertexInputEXT-vertexBindingDescriptionCount-04791**
  `vertexBindingDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-vkCmdSetVertexInputEXT-vertexAttributeDescriptionCount-04792**
  `vertexAttributeDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributes`

- **VUID-vkCmdSetVertexInputEXT-binding-04793**
  For every binding specified by each element of `pVertexAttributeDescriptions`, a `VkVertexInputBindingDescription2EXT` must exist in `pVertexBindingDescriptions` with the same value of binding

- **VUID-vkCmdSetVertexInputEXT-pVertexBindingDescriptions-04794**
  All elements of `pVertexBindingDescriptions` must describe distinct binding numbers

- **VUID-vkCmdSetVertexInputEXT-pVertexAttributeDescriptions-04795**
  All elements of `pVertexAttributeDescriptions` must describe distinct attribute locations

Valid Usage (Implicit)

- **VUID-vkCmdSetVertexInputEXT-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetVertexInputEXT-pVertexBindingDescriptions-parameter**
  If `vertexBindingDescriptionCount` is not 0, `pVertexBindingDescriptions` must be a valid pointer to an array of `vertexBindingDescriptionCount` valid `VkVertexInputBindingDescription2EXT` structures

- **VUID-vkCmdSetVertexInputEXT-pVertexAttributeDescriptions-parameter**
  If `vertexAttributeDescriptionCount` is not 0, `pVertexAttributeDescriptions` must be a valid pointer to an array of `vertexAttributeDescriptionCount` valid `VkVertexInputAttributeDescription2EXT` structures

- **VUID-vkCmdSetVertexInputEXT-commandBuffer-recording**
  `commandBuffer` must be in the recording state

- **VUID-vkCmdSetVertexInputEXT-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkVertexInputBindingDescription2EXT` structure is defined as:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
typedef struct VkVertexInputBindingDescription2EXT {
    VkStructureType      sType;
    void*                pNext;
    uint32_t             binding;
    uint32_t             stride;
    VkVertexInputRate    inputRate;
    uint32_t             divisor;
} VkVertexInputBindingDescription2EXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `binding` is the binding number that this structure describes.
- `stride` is the distance in bytes between two consecutive elements within the buffer.
- `inputRate` is a `VkVertexInputRate` value specifying whether vertex attribute addressing is a function of the vertex index or of the instance index.
- `divisor` is the number of successive instances that will use the same value of the vertex attribute when instanced rendering is enabled. This member can be set to a value other than 1 if the `vertexAttributeInstanceRateDivisor` feature is enabled. For example, if the divisor is N, the same vertex attribute will be applied to N successive instances before moving on to the next vertex attribute. The maximum value of divisor is implementation dependent and can be queried using `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor`. A value of 0 can be used for the divisor if the `vertexAttributeInstanceRateZeroDivisor` feature is enabled. In this case, the same vertex attribute will be applied to all instances.
Valid Usage

- **VUID-VkVertexInputBindingDescription2EXT-binding-04796**
  
  *binding* must be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-VkVertexInputBindingDescription2EXT-stride-04797**
  
  *stride* must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindingStride`

- **VUID-VkVertexInputBindingDescription2EXT-divisor-04798**
  
  If the `vertexAttributeInstanceRateZeroDivisor` feature is not enabled, *divisor* must not be 0

- **VUID-VkVertexInputBindingDescription2EXT-divisor-04799**
  
  If the `vertexAttributeInstanceRateDivisor` feature is not enabled, *divisor* must be 1

- **VUID-VkVertexInputBindingDescription2EXT-divisor-04800**
  
  *divisor* must be a value between 0 and `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor`, inclusive

- **VUID-VkVertexInputBindingDescription2EXT-divisor-04801**
  
  If *divisor* is not 1 then *inputRate* must be of type `VK_VERTEX_INPUT_RATE_INSTANCE`

Valid Usage (Implicit)

- **VUID-VkVertexInputBindingDescription2EXT-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_VERTEX_INPUT_BINDING_DESCRIPTION_2_EXT`

- **VUID-VkVertexInputBindingDescription2EXT-inputRate-parameter**
  
  *inputRate* must be a valid `VkVertexInputRate` value

The `VkVertexInputAttributeDescription2EXT` structure is defined as:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
typedef struct VkVertexInputAttributeDescription2EXT {
    VkStructureType   sType;
    void*             pNext;
    uint32_t           location;
    uint32_t           binding;
    VkFormat           format;
    uint32_t           offset;
} VkVertexInputAttributeDescription2EXT;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *location* is the shader input location number for this attribute.
- *binding* is the binding number which this attribute takes its data from.
- *format* is the size and type of the vertex attribute data.
• **offset** is a byte offset of this attribute relative to the start of an element in the vertex input binding.

**Valid Usage**

- VUID-VkVertexInputAttributeDescription2EXT-location-04802
  location must be less than VkPhysicalDeviceLimits::maxVertexInputAttributes

- VUID-VkVertexInputAttributeDescription2EXT-binding-04803
  binding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings

- VUID-VkVertexInputAttributeDescription2EXT-offset-04804
  offset must be less than or equal to VkPhysicalDeviceLimits::maxVertexInputAttributeOffset

- VUID-VkVertexInputAttributeDescription2EXT-format-04805
  format must be allowed as a vertex buffer format, as specified by the VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT flag in VkFormatProperties::bufferFeatures returned by vkGetPhysicalDeviceFormatProperties

- VUID-VkVertexInputAttributeDescription2EXT-vertexAttributeAccessBeyondStride-04806
  If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::vertexAttributeAccessBeyondStride is VK_FALSE, the sum of offset plus the size of the vertex attribute data described by format must not be greater than stride in the VkVertexInputBindingDescription2EXT referenced in binding.

**Valid Usage (Implicit)**

- VUID-VkVertexInputAttributeDescription2EXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION_2_EXT

- VUID-VkVertexInputAttributeDescription2EXT-format-parameter
  format must be a valid VkFormat value

To bind vertex buffers to a command buffer for use in subsequent draw commands, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdBindVertexBuffers(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstBinding,
    uint32_t                                    bindingCount,
    const VkBuffer*                             pBuffers,
    const VkDeviceSize*                         pOffsets);
```

- **commandBuffer** is the command buffer into which the command is recorded.
- **firstBinding** is the index of the first vertex input binding whose state is updated by the command.
• **bindingCount** is the number of vertex input bindings whose state is updated by the command.
• **pBuffers** is a pointer to an array of buffer handles.
• **pOffsets** is a pointer to an array of buffer offsets.

The values taken from elements i of pBuffers and pOffsets replace the current state for the vertex input binding firstBinding + i, for i in [0, bindingCount). The vertex input binding is updated to start at the offset indicated by pOffsets[i] from the start of the buffer pBuffers[i]. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent draw commands. If the nullDescriptor feature is enabled, elements of pBuffers can be VK_NULL_HANDLE, and can be used by the vertex shader. If a vertex input attribute is bound to a vertex input binding that is VK_NULL_HANDLE, the values taken from memory are considered to be zero, and missing G, B, or A components are filled with (0,0,1).

---

**Valid Usage**

- VUID-vkCmdBindVertexBuffers-firstBinding-00624
  firstBinding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings
- VUID-vkCmdBindVertexBuffers-firstBinding-00625
  The sum of firstBinding and bindingCount must be less than or equal to VkPhysicalDeviceLimits::maxVertexInputBindings
- VUID-vkCmdBindVertexBuffers-pOffsets-00626
  All elements of pOffsets must be less than the size of the corresponding element in pBuffers
- VUID-vkCmdBindVertexBuffers-pBuffers-00627
  All elements of pBuffers must have been created with the VK_BUFFER_USAGE_VERTEX_BUFFER_BIT flag
- VUID-vkCmdBindVertexBuffers-pBuffers-00628
  Each element of pBuffers that is non-sparse must be bound completely and contiguously to a single VkDeviceMemory object
- VUID-vkCmdBindVertexBuffers-pBuffers-04001
  If the nullDescriptor feature is not enabled, all elements of pBuffers must not be VK_NULL_HANDLE
- VUID-vkCmdBindVertexBuffers-pBuffers-04002
  If an element of pBuffers is VK_NULL_HANDLE, then the corresponding element of pOffsets must be zero
Valid Usage (Implicit)

- **VUID-vkCmdBindVertexBuffers-commandBuffer-parameter**
  "commandBuffer must be a valid VkCommandBuffer handle"

- **VUID-vkCmdBindVertexBuffers-pBuffers-parameter**
  "pBuffers must be a valid pointer to an array of bindingCount valid or VK_NULL_HANDLE VkBuffer handles"

- **VUID-vkCmdBindVertexBuffers-pOffsets-parameter**
  "pOffsets must be a valid pointer to an array of bindingCount VkDeviceSize values"

- **VUID-vkCmdBindVertexBuffers-commandBuffer-recording**
  "commandBuffer must be in the recording state"

- **VUID-vkCmdBindVertexBuffers-commandBuffer-cmdpool**
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- **VUID-vkCmdBindVertexBuffers-bindingCount-arraylength**
  "bindingCount must be greater than 0"

- **VUID-vkCmdBindVertexBuffers-commonparent**
  Both of commandBuffer, and the elements of pBuffers that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alternatively, to bind vertex buffers, along with their sizes and strides, to a command buffer for use in subsequent draw commands, call:
// Provided by VK_EXT_extended_dynamic_state

void vkCmdBindVertexBuffers2EXT(
    VkCommandBuffer commandBuffer,
    uint32_t firstBinding,
    uint32_t bindingCount,
    const VkBuffer* pBuffers,
    const VkDeviceSize* pOffsets,
    const VkDeviceSize* pSizes,
    const VkDeviceSize* pStrides);

• `commandBuffer` is the command buffer into which the command is recorded.

• `firstBinding` is the index of the first vertex input binding whose state is updated by the command.

• `bindingCount` is the number of vertex input bindings whose state is updated by the command.

• `pBuffers` is a pointer to an array of buffer handles.

• `pOffsets` is a pointer to an array of buffer offsets.

• `pSizes` is an optional array of the size in bytes of vertex data bound from `pBuffers`.

• `pStrides` is optional, and when not `NULL` is a pointer to an array of buffer strides.

The values taken from elements `i` of `pBuffers` and `pOffsets` replace the current state for the vertex input binding `firstBinding + i`, for `i` in `[0, bindingCount)`. The vertex input binding is updated to start at the offset indicated by `pOffsets[i]` from the start of the buffer `pBuffers[i]`. If `pSizes` is not `NULL` then `pSizes[i]` specifies the bound size of the vertex buffer starting from the corresponding elements of `pBuffers[i]` plus `pOffsets[i]`. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent draw commands.

If the bound pipeline state object was created with the `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT` dynamic state enabled then `pStrides[i]` specifies the distance in bytes between two consecutive elements within the corresponding buffer. In this case the `VkVertexInputBindingDescription::stride` state from the pipeline state object is ignored.
Valid Usage

- **VUID-vkCmdBindVertexBuffers2EXT-firstBinding-03355**
  
  `firstBinding` must be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-vkCmdBindVertexBuffers2EXT-firstBinding-03356**
  
  The sum of `firstBinding` and `bindingCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-vkCmdBindVertexBuffers2EXT-pOffsets-03357**
  
  All elements of `pOffsets` must be less than the size of the corresponding element in `pBuffers`

- **VUID-vkCmdBindVertexBuffers2EXT-pSizes-03358**
  
  If `pSizes` is not `NULL`, all elements of `pOffsets` plus `pSizes` must be less than or equal to the size of the corresponding element in `pBuffers`

- **VUID-vkCmdBindVertexBuffers2EXT-pBuffers-03359**
  
  All elements of `pBuffers` must have been created with the `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` flag

- **VUID-vkCmdBindVertexBuffers2EXT-pBuffers-03360**
  
  Each element of `pBuffers` that is non-sparse must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdBindVertexBuffers2EXT-pBuffers-04111**
  
  If the `nullDescriptor` feature is not enabled, all elements of `pBuffers` must not be `VK_NULL_HANDLE`

- **VUID-vkCmdBindVertexBuffers2EXT-pBuffers-04112**
  
  If an element of `pBuffers` is `VK_NULL_HANDLE`, then the corresponding element of `pOffsets` must be zero

- **VUID-vkCmdBindVertexBuffers2EXT-pStrides-03362**
  
  If `pStrides` is not `NULL` each element of `pStrides` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindingStride`

- **VUID-vkCmdBindVertexBuffers2EXT-pStrides-03363**
  
  If `pStrides` is not `NULL` each element of `pStrides` must be greater than or equal to the maximum extent of of all vertex input attributes fetched from the corresponding binding, where the extent is calculated as the `VkVertexInputAttributeDescription::offset` plus `VkVertexInputAttributeDescription::format` size
Valid Usage (Implicit)

- VUID-vkCmdBindVertexBuffers2EXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBindVertexBuffers2EXT-pBuffers-parameter
  pBuffers must be a valid pointer to an array of bindingCount valid VkBuffer handles

- VUID-vkCmdBindVertexBuffers2EXT-pOffsets-parameter
  pOffsets must be a valid pointer to an array of bindingCount VkDeviceSize values

- VUID-vkCmdBindVertexBuffers2EXT-pSizes-parameter
  If pSizes is not NULL, pSizes must be a valid pointer to an array of bindingCount VkDeviceSize values

- VUID-vkCmdBindVertexBuffers2EXT-pStrides-parameter
  If pStrides is not NULL, pStrides must be a valid pointer to an array of bindingCount VkDeviceSize values

- VUID-vkCmdBindVertexBuffers2EXT-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdBindVertexBuffers2EXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdBindVertexBuffers2EXT-bindingCount-arraylength
  If any of pSizes, or pStrides are not NULL, bindingCount must be greater than 0

- VUID-vkCmdBindVertexBuffers2EXT-commonparent
  Both of commandBuffer, and the elements of pBuffers must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
22.3. Vertex Attribute Divisor in Instanced Rendering

If `vertexAttributeInstanceRateDivisor` feature is enabled and the `pNext` chain of `VkPipelineVertexInputStateCreateInfo` includes a `VkPipelineVertexInputDivisorStateCreateInfoEXT` structure, then that structure controls how vertex attributes are assigned to an instance when instanced rendering is enabled.

The `VkPipelineVertexInputDivisorStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkPipelineVertexInputDivisorStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t vertexBindingDivisorCount;
    const VkVertexInputBindingDivisorDescriptionEXT* pVertexBindingDivisors;
} VkPipelineVertexInputDivisorStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `vertexBindingDivisorCount` is the number of elements in the `pVertexBindingDivisors` array.
- `pVertexBindingDivisors` is a pointer to an array of `VkVertexInputBindingDivisorDescriptionEXT` structures, which specifies the divisor value for each binding.

**Valid Usage (Implicit)**

- `VUID-VkPipelineVertexInputDivisorStateCreateInfoEXT-sType-sType` sType must be `VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT`
- `VUID-VkPipelineVertexInputDivisorStateCreateInfoEXT-pVertexBindingDivisors-parameter` `pVertexBindingDivisors` must be a valid pointer to an array of `vertexBindingDivisorCount` `VkVertexInputBindingDivisorDescriptionEXT` structures
- `VUID-VkPipelineVertexInputDivisorStateCreateInfoEXT-vertexBindingDivisorCount-arraylength` `vertexBindingDivisorCount` must be greater than 0

The individual divisor values per binding are specified using the `VkVertexInputBindingDivisorDescriptionEXT` structure which is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkVertexInputBindingDivisorDescriptionEXT {
    uint32_t binding;
    uint32_t divisor;
} VkVertexInputBindingDivisorDescriptionEXT;
```

- `binding` is the binding number for which the divisor is specified.
• **divisor** is the number of successive instances that will use the same value of the vertex attribute when instanced rendering is enabled. For example, if the divisor is \( N \), the same vertex attribute will be applied to \( N \) successive instances before moving on to the next vertex attribute. The maximum value of divisor is implementation dependent and can be queried using `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor`. A value of 0 can be used for the divisor if the `vertexAttributeInstanceRateZeroDivisor` feature is enabled. In this case, the same vertex attribute will be applied to all instances.

If this structure is not used to define a divisor value for an attribute then the divisor has a logical default value of 1.

### Valid Usage

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-binding-01869**
  
  binding must be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-vertexAttributeInstanceRateZeroDivisor-02228**
  
  If the `vertexAttributeInstanceRateZeroDivisor` feature is not enabled, divisor must not be 0

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-vertexAttributeInstanceRateDivisor-02229**
  
  If the `vertexAttributeInstanceRateDivisor` feature is not enabled, divisor must be 1

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-divisor-01870**
  
  divisor must be a value between 0 and `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT::maxVertexAttribDivisor`, inclusive

- **VUID-VkVertexInputBindingDivisorDescriptionEXT-inputRate-01871**
  
  `VkVertexInputBindingDescription::inputRate` must be of type `VK_VERTEX_INPUT_RATE_INSTANCE` for this binding

The address of each attribute for each `vertexIndex` and `instanceIndex` is calculated as follows:

- Let `attribDesc` be the member of `VkPipelineVertexInputStateCreateInfo::pVertexAttributeDescriptions` with `VkVertexInputAttributeDescription::location` equal to the vertex input attribute number.

- Let `bindingDesc` be the member of `VkPipelineVertexInputStateCreateInfo::pVertexBindingDescriptions` with `VkVertexInputAttributeDescription::binding` equal to `attribDesc.binding`.

- Let `vertexIndex` be the index of the vertex within the draw (a value between `firstVertex` and `firstVertex+vertexCount` for `vkCmdDraw`, or a value taken from the index buffer for `vkCmdDrawIndexed`), and let `instanceIndex` be the instance number of the draw (a value between `firstInstance` and `firstInstance+instanceCount`).

- Let `divisor` be the member of `VkPipelineVertexInputDivisorStateCreateInfoEXT::pVertexBindingDivisors` with `VkVertexInputBindingDivisorDescriptionEXT::binding` equal to `attribDesc.binding`. 
bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    vertexOffset = vertexIndex * bindingDesc.stride;
else
    if (divisor == 0)
        vertexOffset = firstInstance * bindingDesc.stride;
    else
        vertexOffset = (firstInstance + ((instanceIndex - firstInstance) / divisor)) * bindingDesc.stride;

attribAddress = bufferBindingAddress + vertexOffset + attribDesc.offset;

For each attribute, raw data is extracted starting at attribAddress and is converted from the VkVertexInputAttributeDescription's format to either floating-point, unsigned integer, or signed integer based on the base type of the format; the base type of the format must match the base type of the input variable in the shader. If format is a packed format, attribAddress must be a multiple of the size in bytes of the whole attribute data type as described in Packed Formats. Otherwise, attribAddress must be a multiple of the size in bytes of the component type indicated by format (see Formats). If the format does not include G, B, or A components, then those are filled with (0,0,1) as needed (using either 1.0f or integer 1 based on the format) for attributes that are not 64-bit data types. The number of components in the vertex shader input variable need not exactly match the number of components in the format. If the vertex shader has fewer components, the extra components are discarded.
Chapter 23. Tessellation

Tessellation involves three pipeline stages. First, a tessellation control shader transforms control points of a patch and can produce per-patch data. Second, a fixed-function tessellator generates multiple primitives corresponding to a tessellation of the patch in (u,v) or (u,v,w) parameter space. Third, a tessellation evaluation shader transforms the vertices of the tessellated patch, for example to compute their positions and attributes as part of the tessellated surface. The tessellator is enabled when the pipeline contains both a tessellation control shader and a tessellation evaluation shader.

23.1. Tessellator

If a pipeline includes both tessellation shaders (control and evaluation), the tessellator consumes each input patch (after vertex shading) and produces a new set of independent primitives (points, lines, or triangles). These primitives are logically produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch outer and inner tessellation levels written by the tessellation control shader. These levels are specified using the built-in variables TessLevelOuter and TessLevelInner, respectively. This subdivision is performed in an implementation-dependent manner. If no tessellation shaders are present in the pipeline, the tessellator is disabled and incoming primitives are passed through without modification.

The type of subdivision performed by the tessellator is specified by an OpExecutionMode instruction in the tessellation evaluation or tessellation control shader using one of execution modes Triangles, Quads, and IsoLines. Other tessellation-related execution modes can also be specified in either the tessellation control or tessellation evaluation shaders, and if they are specified in both then the modes must be the same.

Tessellation execution modes include:

- **Triangles, Quads, and IsoLines.** These control the type of subdivision and topology of the output primitives. One mode must be set in at least one of the tessellation shader stages. If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::tessellationIsolines is VK_FALSE, then isoline tessellation is not supported by the implementation, and IsoLines must not be used in either tessellation shader stage.

- **VertexOrderCw and VertexOrderCcw.** These control the orientation of triangles generated by the tessellator. One mode must be set in at least one of the tessellation shader stages.

- **PointMode.** Controls generation of points rather than triangles or lines. This functionality defaults to disabled, and is enabled if either shader stage includes the execution mode. If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::tessellationPointMode is VK_FALSE, then point mode tessellation is not supported by the implementation, and PointMode must not be used in either tessellation shader stage.

- **SpacingEqual, SpacingFractionalEven, and SpacingFractionalOdd.** Controls the spacing of segments on the edges of tessellated primitives. One mode must be set in at least one of the tessellation shader stages.
• **OutputVertices.** Controls the size of the output patch of the tessellation control shader. One value **must** be set in at least one of the tessellation shader stages.

For triangles, the tessellator subdivides a triangle primitive into smaller triangles. For quads, the tessellator subdivides a rectangle primitive into smaller triangles. For isolines, the tessellator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching across the rectangle in the u dimension (i.e. the coordinates in `TessCoord` are of the form $(0,x)$ through $(1,x)$ for all tessellation evaluation shader invocations that share a line).

Each vertex produced by the tessellator has an associated $(u,v,w)$ or $(u,v)$ position in a normalized parameter space, with parameter values in the range $[0,1]$, as illustrated in figures **Domain parameterization for tessellation primitive modes (upper-left origin)** and **Domain parameterization for tessellation primitive modes (lower-left origin)**. The domain space **can** have either an upper-left or lower-left origin, selected by the `domainOrigin` member of `VkPipelineTessellationDomainOriginStateCreateInfo`.

![Domain parameterization](image)

*Figure 12. Domain parameterization for tessellation primitive modes (upper-left origin)*
Figure 13. Domain parameterization for tessellation primitive modes (lower-left origin)

Caption
In the domain parameterization diagrams, the coordinates illustrate the value of `TessCoord` at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

For triangles, the vertex's position is a barycentric coordinate \((u,v,w)\), where \(u + v + w = 1.0\), and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a \((u,v)\) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

23.2. Tessellator Patch Discard

A patch is discarded by the tessellator if any relevant outer tessellation level is less than or equal to zero.

Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point NaN (not a number) in implementations supporting NaN.

No new primitives are generated and the tessellation evaluation shader is not executed for patches that are discarded. For Quads, all four outer levels are relevant. For Triangles and Isolines, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.
23.3. Tessellator Spacing

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an `OpExecutionMode` in the tessellation control or tessellation evaluation shader using one of the identifiers `SpacingEqual`, `SpacingFractionalEven`, or `SpacingFractionalOdd`.

If `SpacingEqual` is used, the floating-point tessellation level is first clamped to \([1, \text{maxLevel}]\), where `maxLevel` is the implementation-dependent maximum tessellation level (`VkPhysicalDeviceLimits::maxTessellationGenerationLevel`). The result is rounded up to the nearest integer `n`, and the corresponding edge is divided into `n` segments of equal length in \((u,v)\) space.

If `SpacingFractionalEven` is used, the tessellation level is first clamped to \([2, \text{maxLevel}]\) and then rounded up to the nearest even integer `n`. If `SpacingFractionalOdd` is used, the tessellation level is clamped to \([1, \text{maxLevel} - 1]\) and then rounded up to the nearest odd integer `n`. If `n` is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into \(n - 2\) segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with \(n - f\), where `f` is the clamped floating-point tessellation level. When \(n - f\) is zero, the additional segments will have equal length to the other segments. As \(n - f\) approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments must be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is implementation-dependent, but must be identical for any pair of subdivided edges with identical values of `f`.

When tessellating triangles or quads using point mode with fractional odd spacing, the tessellator may produce interior vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

23.4. Tessellation Primitive Ordering

Few guarantees are provided for the relative ordering of primitives produced by tessellation, as they pertain to primitive order.

- The output primitives generated from each input primitive are passed to subsequent pipeline stages in an implementation-dependent order.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

23.5. Tessellator Vertex Winding Order

When the tessellator produces triangles (in the Triangles or Quads modes), the orientation of all triangles is specified with an `OpExecutionMode` of `VertexOrderCw` or `VertexOrderCcw` in the tessellation control or tessellation evaluation shaders. If the order is `VertexOrderCw`, the vertices of all generated triangles will have clockwise ordering in \((u,v)\) or \((u,v,w)\) space. If the order is `VertexOrderCcw`, the
vertices will have counter-clockwise ordering in that space.

If the tessellation domain has an upper-left origin, the vertices of a triangle have counter-clockwise ordering if

\[ a = u_0 v_1 - u_1 v_0 + u_1 v_2 - u_2 v_1 + u_2 v_0 - u_0 v_2 \]

is negative, and clockwise ordering if \( a \) is positive. \( u_i \) and \( v_i \) are the \( u \) and \( v \) coordinates in normalized parameter space of the \( i \)th vertex of the triangle. If the tessellation domain has a lower-left origin, the vertices of a triangle have counter-clockwise ordering if \( a \) is positive, and clockwise ordering if \( a \) is negative.

**Note**

The value \( a \) is proportional (with a positive factor) to the signed area of the triangle.

In *Triangles* mode, even though the vertex coordinates have a \( w \) value, it does not participate directly in the computation of \( a \), being an affine combination of \( u \) and \( v \).

### 23.6. Triangle Tessellation

If the tessellation primitive mode is *Triangles*, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the \( u = 0 \) (left), \( v = 0 \) (bottom), and \( w = 0 \) (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with \((u,v,w)\) coordinates of \((0,0,1)\), \((1,0,0)\), and \((0,1,0)\) is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as \(1 + \varepsilon\) and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce inner vertices positioned on the edge of the triangle.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating \( n \) segments. For the outermost inner triangle, the inner triangle is degenerate — a single point at the center of the triangle — if \( n \) is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the
corner's two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If \( n \) is three, the edges of the inner triangle are not subdivided and it is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into \( n - 2 \) segments, with the \( n - 1 \) vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the \( n - 1 \) innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in Inner Triangle Tessellation.

![Inner Triangle Tessellation](image)

**Figure 14. Inner Triangle Tessellation**

**Caption**

In the Inner Triangle Tessellation diagram, inner tessellation levels of (a) five and (b) four are shown (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex of the subdivided outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.

After the area corresponding to any inner triangles is filled, the tessellator generates triangles to cover the area between the outermost triangle and the outermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the \( u = 0 \), \( v = 0 \), and \( w = 0 \) edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.
After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric 
(u,v,w) coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in (u,v,w) space into individual triangles is 
implementation-dependent. However, the set of triangles produced will completely cover the 
domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to triangle lists, except that the order in 
which each triangle is generated, and the order in which the vertices are generated for each 
triangle, are implementation-dependent. However, the order of vertices in each triangle is 
consistent across the domain as described in Tessellator Vertex Winding Order.

23.7. Quad Tessellation

If the tessellation primitive mode is Quads, a rectangle is subdivided into a collection of triangles 
covering the area of the original rectangle. First, the original rectangle is subdivided into a regular 
grid of rectangles, where the number of rectangles along the u = 0 and u = 1 (vertical) and v = 0 
and v = 1 (horizontal) edges are derived from the first and second inner tessellation levels, 
respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are 
decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using 
the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of 
the u = 0 (left), v = 0 (bottom), u = 1 (right), and v = 1 (top) edges, respectively. The area between the 
inner rectangles of the mesh and the outer rectangle edges are filled by triangles produced by 
joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle 
mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly 
one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either 
clamped inner tessellation level is one, that tessellation level is treated as though it was originally 
specified as 1 + ε and will result in a two- or three-segment subdivision depending on the 
tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may 
produce inner vertices positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the u = 0 and u = 1 
edges of the outer rectangle into m segments using the clamped and rounded first inner tessellation 
level and the tessellation spacing. The v = 0 and v = 1 edges are subdivided into n segments using 
the second inner tessellation level. Each vertex on the u = 0 and v = 0 edges are joined with the 
corresponding vertex on the u = 1 and v = 1 edges to produce a set of vertical and horizontal lines 
that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of 
non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer 
rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the 
edges of which are subdivided by the grid vertices that lie on the edge. If either m or n is two, the 
inner rectangle is degenerate, and one or both of the rectangle’s edges consist of a single point. This 
subdivision is illustrated in Figure Inner Quad Tessellation.
In the Inner Quad Tessellation diagram, inner quad tessellation levels of (a) (4,2) and (b) (7,4) are shown. The regions highlighted in red in figure (b) depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle of figure (a) is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and outer rectangle edges.

After the area corresponding to the inner rectangle is filled, the tessellator must produce triangles to cover the area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edge above is discarded. Instead, the u = 0, v = 0, u = 1, and v = 1 edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other rectangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edges is filled by connecting each vertex on the outer edge with the single vertex making up the inner edge.

The algorithm used to subdivide the rectangular domain in (u,v) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles.

Output triangles are generated with a topology similar to triangle lists, except that the order in which each triangle is generated, and the order in which the vertices are generated for each triangle, are implementation-dependent. However, the order of vertices in each triangle is consistent across the domain as described in Tessellator Vertex Winding Order.

23.8. Isoline Tessellation

If the tessellation primitive mode is Isolines, a set of independent horizontal line segments is drawn. The segments are arranged into connected strips called isolines, where the vertices of each isoline have a constant v coordinate and u coordinates covering the full range [0,1]. The number of
isolines generated is derived from the first outer tessellation level; the number of segments in each isoline is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The \( u = 0 \) and \( u = 1 \) edges of the rectangle are subdivided according to the first outer tessellation level. For the purposes of this subdivision, the tessellation spacing mode is ignored and treated as equal_spacing. An isoline is drawn connecting each vertex on the \( u = 0 \) rectangle edge to the corresponding vertex on the \( u = 1 \) rectangle edge, except that no line is drawn between \((0,1)\) and \((1,1)\). If the number of isolines on the subdivided \( u = 0 \) and \( u = 1 \) edges is \( n \), this process will result in \( n \) equally spaced lines with constant \( v \) coordinates of \( 0, \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n} \).

Each of the \( n \) isolines is then subdivided according to the second outer tessellation level and the tessellation spacing, resulting in \( m \) line segments. Each segment of each line is emitted by the tessellator. These line segments are generated with a topology similar to line lists, except that the order in which each line is generated, and the order in which the vertices are generated for each line segment, are implementation-dependent.

**Note**

If the \texttt{VK_KHR_portability_subset} extension is enabled, and \texttt{VkPhysicalDevicePortabilitySubsetFeaturesKHR::tessellationIsolines} is \texttt{VK_FALSE}, then isoline tessellation is not supported by the implementation.

### 23.9. Tessellation Point Mode

For all primitive modes, the tessellator is capable of generating points instead of lines or triangles. If the tessellation control or tessellation evaluation shader specifies the \texttt{OpExecutionMode PointMode}, the primitive generator will generate one point for each distinct vertex produced by tessellation, rather than emitting triangles or lines. Otherwise, the tessellator will produce a collection of line segments or triangles according to the primitive mode. These points are generated with a topology similar to point lists, except the order in which the points are generated for each input primitive is undefined.

**Note**

If the \texttt{VK_KHR_portability_subset} extension is enabled, and \texttt{VkPhysicalDevicePortabilitySubsetFeaturesKHR::tessellationPointMode} is \texttt{VK_FALSE}, then tessellation point mode is not supported by the implementation.

### 23.10. Tessellation Pipeline State

The \texttt{pTessellationState} member of \texttt{VkGraphicsPipelineCreateInfo} is a pointer to a \texttt{VkPipelineTessellationStateCreateInfo} structure.

The \texttt{VkPipelineTessellationStateCreateInfo} structure is defined as:
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineTessellationStateCreateFlags flags;
    uint32_t patchControlPoints;
} VkPipelineTessellationStateCreateInfo;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **patchControlPoints** is the number of control points per patch.

### Valid Usage

- VUID-VkPipelineTessellationStateCreateInfo-patchControlPoints-01214
  - **patchControlPoints** must be greater than zero and less than or equal to
    VkPhysicalDeviceLimits::maxTessellationPatchSize

### Valid Usage (Implicit)

- VUID-VkPipelineTessellationStateCreateInfo-sType-sType
  - **sType** must be VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO

- VUID-VkPipelineTessellationStateCreateInfo-pNext-pNext
  - **pNext** must be NULL or a pointer to a valid instance of
    VkPipelineTessellationDomainOriginStateCreateInfo

- VUID-VkPipelineTessellationStateCreateInfo-sType-unique
  - The **sType** value of each struct in the **pNext** chain must be unique

- VUID-VkPipelineTessellationStateCreateInfo-flags-zerobitmask
  - **flags** must be 0

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineTessellationStateCreateFlags;

**VkPipelineTessellationStateCreateFlags** is a bitmask type for setting a mask, but is currently
reserved for future use.

The **VkPipelineTessellationDomainOriginStateCreateInfo** structure is defined as:
```c
// Provided by VK_VERSION_1_1
typedef struct VkPipelineTessellationDomainOriginStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkTessellationDomainOrigin domainOrigin;
} VkPipelineTessellationDomainOriginStateCreateInfo;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance2
typedef VkPipelineTessellationDomainOriginStateCreateInfo
VkPipelineTessellationDomainOriginStateCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `domainOrigin` is a `VkTessellationDomainOrigin` value controlling the origin of the tessellation domain space.

If the `VkPipelineTessellationDomainOriginStateCreateInfo` structure is included in the `pNext` chain of `VkPipelineTessellationStateCreateInfo`, it controls the origin of the tessellation domain. If this structure is not present, it is as if `domainOrigin` was `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT`.

### Valid Usage (Implicit)

- VUID-VkPipelineTessellationDomainOriginStateCreateInfo-sType-sType
  - `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO`

- VUID-VkPipelineTessellationDomainOriginStateCreateInfo-domainOrigin-parameter
  - `domainOrigin` **must** be a valid `VkTessellationDomainOrigin` value

The possible tessellation domain origins are specified by the `VkTessellationDomainOrigin` enumeration:

```c
// Provided by VK_VERSION_1_1
typedef enum VkTessellationDomainOrigin {
    VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT = 0,
    VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT = 1,
    // Provided by VK_KHR_maintenance2
    VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT_KHR = VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT,
    // Provided by VK_KHR_maintenance2
    VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT_KHR = VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT,
} VkTessellationDomainOrigin;
```

1436 | Chapter 23. Tessellation
or the equivalent

```c
// Provided by VK_KHR_maintenance2
typedef VkTessellationDomainOrigin VkTessellationDomainOriginKHR;
```

- **VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT** specifies that the origin of the domain space is in the upper left corner, as shown in figure Domain parameterization for tessellation primitive modes (upper-left origin).

- **VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT** specifies that the origin of the domain space is in the lower left corner, as shown in figure Domain parameterization for tessellation primitive modes (lower-left origin).

This enum affects how the `VertexOrderCw` and `VertexOrderCcw` tessellation execution modes are interpreted, since the winding is defined relative to the orientation of the domain.
Chapter 24. Geometry Shading

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive. Geometry shading is enabled when a geometry shader is included in the pipeline.

24.1. Geometry Shader Input Primitives

Each geometry shader invocation has access to all vertices in the primitive (and their associated data), which are presented to the shader as an array of inputs.

The input primitive type expected by the geometry shader is specified with an `OpExecutionMode` instruction in the geometry shader, and must match the incoming primitive type specified by either the pipeline's primitive topology if tessellation is inactive, or the tessellation mode if tessellation is active, as follows:

- An input primitive type of `InputPoints` must only be used with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_POINT_LIST`, or with a tessellation shader that specifies `PointMode`. The input arrays always contain one element, as described by the point list topology or tessellation in point mode.

- An input primitive type of `InputLines` must only be used with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` or `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`, or with a tessellation shader specifying `IsoLines` that does not specify `PointMode`. The input arrays always contain two elements, as described by the line list topology or line strip topology, or by isoline tessellation.

- An input primitive type of `InputLinesAdjacency` must only be used when tessellation is inactive, with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`. The input arrays always contain four elements, as described by the line list with adjacency topology or line strip with adjacency topology.

- An input primitive type of `Triangles` must only be used with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`, `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`, or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`; or with a tessellation shader specifying `Quads` or `Triangles` that does not specify `PointMode`. The input arrays always contain three elements, as described by the triangle list topology, triangle strip topology, or triangle fan topology, or by triangle or quad tessellation. Vertices may be in a different absolute order to that specified by the topology, but must adhere to the specified winding order.

- An input primitive type of `InputTrianglesAdjacency` must only be used when tessellation is inactive, with a pipeline topology of `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY` or `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`. The input arrays always contain six elements, as described by the triangle list with adjacency topology or triangle strip with adjacency topology. Vertices may be in a different absolute order to that specified by the topology, but must adhere to the specified winding order, and the vertices making up the main primitive must still occur at the first, third, and fifth index.
24.2. Geometry Shader Output Primitives

A geometry shader generates primitives in one of three output modes: points, line strips, or triangle strips. The primitive mode is specified in the shader using an `OpExecutionMode` instruction with the `OutputPoints`, `OutputLineStrip` or `OutputTriangleStrip` modes, respectively. Each geometry shader **must** include exactly one output primitive mode.

The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type and the resulting primitives are then further processed as described in Rasterization. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, vertices corresponding to incomplete primitives are not processed by subsequent pipeline stages. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The maximum output vertex count is specified in the shader using an `OpExecutionMode` instruction with the mode set to `OutputVertices` and the maximum number of vertices that will be produced by the geometry shader specified as a literal. Each geometry shader **must** specify a maximum output vertex count.

24.3. Multiple Invocations of Geometry Shaders

Geometry shaders **can** be invoked more than one time for each input primitive. This is known as *geometry shader instancing* and is requested by including an `OpExecutionMode` instruction with mode specified as `Invocations` and the number of invocations specified as an integer literal.

In this mode, the geometry shader will execute at least \( n \) times for each input primitive, where \( n \) is the number of invocations specified in the `OpExecutionMode` instruction. The instance number is available to each invocation as a built-in input using `InvocationId`.

24.4. Geometry Shader Primitive Ordering

Limited guarantees are provided for the relative ordering of primitives produced by a geometry shader, as they pertain to **primitive order**.

- For instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the invocation number to order the primitives, from least to greatest.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

24.5. Geometry Shader Passthrough

A geometry shader that uses the `PassthroughNV` decoration on a variable in its input interface is considered a *passthrough geometry shader*. Output primitives in a passthrough geometry shader **must** have the same topology as the input primitive and are not produced by emitting vertices. The vertices of the output primitive have two different types of attributes, per-vertex and per-primitive. Geometry shader input variables with `PassthroughNV` decoration are considered to produce per-
vertex outputs, where values for each output vertex are copied from the corresponding input vertex. Any built-in or user-defined geometry shader outputs are considered per-primitive in a passthrough geometry shader, where a single output value is copied to all output vertices.

The remainder of this section details the usage of the \texttt{PassthroughNV} decoration and modifications to the interface matching rules when using passthrough geometry shaders.

\subsection*{24.5.1. \texttt{PassthroughNV} Decoration}

Decorating a geometry shader input variable with the \texttt{PassthroughNV} decoration indicates that values of this input are copied through to the corresponding vertex of the output primitive. Input variables and block members which do not have the \texttt{PassthroughNV} decoration are consumed by the geometry shader without being passed through to subsequent stages.

The \texttt{PassthroughNV} decoration \textbf{must} only be used within a geometry shader.

Any variable decorated with \texttt{PassthroughNV} \textbf{must} be declared using the \texttt{Input} storage class.

The \texttt{PassthroughNV} decoration \textbf{must} not be used with any of:

- an input primitive type other than \texttt{InputPoints}, \texttt{InputLines}, or \texttt{Triangles}, as specified by the \texttt{mode} for \texttt{OpExecutionMode}.
- an invocation count other than one, as specified by the \texttt{Invocations} mode for \texttt{OpExecutionMode}.
- an \texttt{OpEntryPoint} which statically uses the \texttt{Op EmitVertex} or \texttt{OpEndPrimitive} instructions.
- a variable decorated with the \texttt{InvocationId} built-in decoration.
- a variable decorated with the \texttt{PrimitiveId} built-in decoration that is declared using the \texttt{Input} storage class.

\subsection*{24.5.2. Passthrough Interface Matching}

When a passthrough geometry shader is in use, the \texttt{Interface Matching} rules involving the geometry shader input and output interfaces operate as described in this section.

For the purposes of matching passthrough geometry shader inputs with outputs of the previous pipeline stages, the \texttt{PassthroughNV} decoration is ignored.

For the purposes of matching the outputs of the geometry shader with subsequent pipeline stages, each input variable with the \texttt{PassthroughNV} decoration is considered to add an equivalent output variable with the same type, decoration (other than \texttt{PassthroughNV}), number, and declaration order on the output interface. The output variable declaration corresponding to an input variable decorated with \texttt{PassthroughNV} will be identical to the input declaration, except that the outermost array dimension of such variables is removed. The output block declaration corresponding to an input block decorated with \texttt{PassthroughNV} or having members decorated with \texttt{PassthroughNV} will be identical to the input declaration, except that the outermost array dimension of such declaration is removed.

If an input block is decorated with \texttt{PassthroughNV}, the equivalent output block contains all the members of the input block. Otherwise, the equivalent output block contains only those input block
members decorated with `PassthroughNV`. All members of the corresponding output block are assigned `Location` and `Component` decorations identical to those assigned to the corresponding input block members.

Output variables and blocks generated from inputs decorated with `PassthroughNV` will only exist for the purposes of interface matching; these declarations are not available to geometry shader code or listed in the module interface.

For the purposes of component counting, passthrough geometry shaders count all statically used input variable components declared with the `PassthroughNV` decoration as output components as well, since their values will be copied to the output primitive produced by the geometry shader.
Chapter 25. Mesh Shading

Task and mesh shaders operate in workgroups to produce a collection of primitives that will be processed by subsequent stages of the graphics pipeline.

Work on the mesh pipeline is initiated by the application drawing a set of mesh tasks organized in global workgroups. If the optional task shader is active, each workgroup triggers the execution of task shader invocations that will create a new set of mesh workgroups upon completion. Each of these created workgroups, or each of the original workgroups if no task shader is present, triggers the execution of mesh shader invocations.

Each mesh shader workgroup emits zero or more output primitives along with the group of vertices and their associated data required for each output primitive.

25.1. Task Shader Input

For every workgroup issued via the drawing commands a group of task shader invocations is executed. There are no inputs other than the builtin workgroup identifiers.

25.2. Task Shader Output

The task shader can emit zero or more mesh workgroups to be generated using the built-in variable TaskCountNV. This value must be less than or equal to VkPhysicalDeviceMeshShaderPropertiesNV::maxTaskOutputCount.

It can also output user-defined data that is passed as input to all mesh shader invocations that the task creates. These outputs are decorated as PerTaskNV.

25.3. Mesh Generation

If a task shader exists, the mesh assembler creates a variable amount of mesh workgroups depending on each task’s output. If there is no task shader, the drawing commands emit the mesh shader invocations directly.

25.4. Mesh Shader Input

The only inputs available to the mesh shader are variables identifying the specific workgroup and invocation and, if applicable, any outputs written as PerTaskNV by the task shader that spawned the mesh shader’s workgroup. The mesh shader can operate without a task shader as well.

25.5. Mesh Shader Output Primitives

A mesh shader generates primitives in one of three output modes: points, lines, or triangles. The primitive mode is specified in the shader using an OpExecutionMode instruction with the OutputPoints, OutputLinesNV, or OutputTrianglesNV modes, respectively. Each mesh shader must include exactly one output primitive mode.
The maximum output vertex count is specified as a literal in the shader using an `OpExecutionMode` instruction with the mode set to `OutputVertices` and must be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputVertices`.

The maximum output primitive count is specified as a literal in the shader using an `OpExecutionMode` instruction with the mode set to `OutputPrimitivesNV` and must be less than or equal to `VkPhysicalDeviceMeshShaderPropertiesNV::maxMeshOutputPrimitives`.

The number of primitives output by the mesh shader is provided via writing to the built-in variable `PrimitiveCountNV` and must be less than or equal to the maximum output primitive count specified in the shader. A variable decorated with `PrimitiveIndicesNV` is an output array of local index values into the vertex output arrays from which primitives are assembled according to the output primitive type. These resulting primitives are then further processed as described in `Rasterization`.

### 25.6. Mesh Shader Per-View Outputs

The mesh shader outputs decorated with the `PositionPerViewNV`, `ClipDistancePerViewNV`, `CullDistancePerViewNV`, `LayerPerViewNV`, and `ViewportMaskPerViewNV` built-in decorations are the per-view versions of the single-view variables with equivalent names (that is `Position`, `ClipDistance`, `CullDistance`, `Layer`, and `ViewportMaskNV`, respectively). If a shader statically assigns a value to any element of a per-view array it must not statically assign a value to the equivalent single-view variable.

Each of these outputs is considered arrayed, with separate values for each view. The view number is used to index the first dimension of these arrays.

The second dimension of the `ClipDistancePerViewNV`, and `CullDistancePerViewNV` arrays have the same requirements as the `ClipDistance`, and `CullDistance` arrays.

If a mesh shader output is `per-view`, the corresponding fragment shader input is taken from the element of the per-view output array that corresponds to the view that is currently being processed by the fragment shader.

### 25.7. Mesh Shader Primitive Ordering

Following guarantees are provided for the relative ordering of primitives produced by a mesh shader, as they pertain to primitive order.

- When a task shader is used, mesh workgroups spawned from lower tasks will be ordered prior to those workgroups from subsequent tasks.
- All output primitives generated from a given mesh workgroup are passed to subsequent pipeline stages before any output primitives generated from subsequent input workgroups.
- All output primitives within a mesh workgroup, will be generated in the ordering provided by the built-in primitive index buffer (from low address to high address).
Chapter 26. Fixed-Function Vertex Post-Processing

After programmable vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

• Transform feedback (see Transform Feedback)
• Viewport swizzle (see Viewport Swizzle)
• Flat shading (see Flat Shading).
• Primitive clipping, including client-defined half-spaces (see Primitive Clipping).
• Shader output attribute clipping (see Clipping Shader Outputs).
• Clip space W scaling (see Controlling Viewport W Scaling).
• Perspective division on clip coordinates (see Coordinate Transformations).
• Viewport mapping, including depth range scaling (see Controlling the Viewport).
• Front face determination for polygon primitives (see Basic Polygon Rasterization).

Next, rasterization is performed on primitives as described in chapter Rasterization.

26.1. Transform Feedback

Before any other fixed-function vertex post-processing, vertex outputs from the last shader in the vertex processing stage can be written out to one or more transform feedback buffers bound to the command buffer. To capture vertex outputs the last vertex processing stage shader must be declared with the Xfb execution mode. Outputs decorated with XfbBuffer will be written out to the corresponding transform feedback buffers bound to the command buffer when transform feedback is active. Transform feedback buffers are bound to the command buffer by using vkCmdBindTransformFeedbackBuffersEXT. Transform feedback is made active by calling vkCmdBeginTransformFeedbackEXT and made inactive by calling vkCmdEndTransformFeedbackEXT. After vertex data is written it is possible to use vkCmdDrawIndirectByteCountEXT to start a new draw where the vertexCount is derived from the number of bytes written by a previous transform feedback.

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active, the values of the specified output variables are assembled into primitives and appended to the bound transform feedback buffers. After activating transform feedback, the values of the first assembled primitive are written at the starting offsets of the bound transform feedback buffers, and subsequent primitives are appended to the buffer. If the optional pCounterBuffers and pCounterBufferOffsets parameters are specified, the starting points within the transform feedback buffers are adjusted so data is appended to the previously written values indicated by the value stored by the implementation in the counter buffer.

For multi-vertex primitives, all values for a given vertex are written before writing values for any other vertex. When transformFeedbackPreservesProvokingVertex is not enabled, implementations may write out any vertex within the primitive first, but all subsequent vertices for that primitive
must be written out in a consistent winding order defined as follows:

- If neither geometry or tessellation shading is active, vertices within a primitive are appended according to the winding order described by the primitive topology defined by the `VkPipelineInputAssemblyStateCreateInfo::topology` used to execute the drawing command.

- If geometry shading is active, vertices within a primitive are appended according to the winding order described by the primitive topology defined by the `OutputPoints`, `OutputLineStrips`, or `OutputTriangleStrips` execution mode.

- If tessellation shading is active but geometry shading is not, vertices within a primitive are appended according to the winding order defined by triangle tessellation, quad tessellation, and isoline tessellation.

When `transformFeedbackPreservesProvokingVertex` is enabled, then in addition to writing vertices with a consistent winding order, the vertex order must preserve the provoking vertex of each primitive:

- When the pipeline's provoking vertex mode is `VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT`, the primitive's provoking vertex must be the first vertex written.

- When the pipeline's provoking vertex mode is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the primitive's provoking vertex must be the last vertex written.

If `transformFeedbackPreservesTriangleFanProvokingVertex` is `VK_FALSE`, neither geometry nor tessellation shading is active, and the primitive topology is `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`, then the first vertex written from each primitive is implementation-defined even when `transformFeedbackPreservesProvokingVertex` is enabled.

When capturing vertices, the stride associated with each transform feedback buffer, as indicated by the `XfbStride` decoration, indicates the number of bytes of storage reserved for each vertex in the transform feedback buffer. For every vertex captured, each output attribute with a `Offset` decoration will be written to the storage reserved for the vertex at the associated transform feedback buffer. When writing output variables that are arrays or structures, individual array elements or structure members are written tightly packed in order. For vector types, individual components are written in order. For matrix types, outputs are written as an array of column vectors.

If any component of an output with an assigned transform feedback offset was not written to by its shader, the value recorded for that component is undefined. All components of an output variable must be written at an offset aligned to the size of the component. The size of each component of an output variable must be at least 32-bits. When capturing a vertex, any portion of the reserved storage not associated with an output variable with an assigned transform feedback offset will be unmodified.

When transform feedback is inactive, no vertices are recorded. If there is a valid counter buffer handle and counter buffer offset in the `pCounterBuffers` and `pCounterBufferOffsets` arrays, writes to the corresponding transform feedback buffer will start at the byte offset represented by the value stored in the counter buffer location.

Individual lines or triangles of a strip or fan primitive will be extracted and recorded separately.
Incomplete primitives are not recorded.

When using a geometry shader that emits vertices to multiple vertex streams, a primitive will be assembled and output for each stream when there are enough vertices emitted for the output primitive type. All outputs assigned to a given transform feedback buffer are required to come from a single vertex stream.

The sizes of the transform feedback buffers are defined by the `vkCmdBindTransformFeedbackBuffersEXT` parameter for each of the bound buffers, or the size of the bound buffer, whichever is the lesser. If there is less space remaining in any of the transform feedback buffers than the size of the all the vertex data for that primitive based on the `XfbStride` for that `XfbBuffer` then no vertex data of that primitive is recorded in any transform feedback buffer, and the value for the number of primitives written in the corresponding `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT` query for all transform feedback buffers is no longer incremented.

Any outputs made to a `XfbBuffer` that is not bound to a transform feedback buffer is ignored.

To bind transform feedback buffers to a command buffer for use in subsequent draw commands, call:

```c
// Provided by VK_EXT_transform_feedback
void vkCmdBindTransformFeedbackBuffersEXT(
    VkCommandBuffer commandBuffer,  // Provided by VK_EXT_transform_feedback
    uint32_t firstBinding,         // Provided by VK_EXT_transform_feedback
    uint32_t bindingCount,         // Provided by VK_EXT_transform_feedback
    const VkBuffer* pBuffers,      // Provided by VK_EXT_transform_feedback
    const VkDeviceSize* pOffsets,  // Provided by VK_EXT_transform_feedback
    const VkDeviceSize* pSizes);   // Provided by VK_EXT_transform_feedback
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `firstBinding` is the index of the first transform feedback binding whose state is updated by the command.
- `bindingCount` is the number of transform feedback bindings whose state is updated by the command.
- `pBuffers` is a pointer to an array of buffer handles.
- `pOffsets` is a pointer to an array of buffer offsets.
- `pSizes` is an optional array of buffer sizes, specifying the maximum number of bytes to capture to the corresponding transform feedback buffer. If `pSizes` is `NULL`, or the value of the `pSizes` array element is `VK_WHOLE_SIZE`, then the maximum bytes captured will be the size of the corresponding buffer minus the buffer offset.

The values taken from elements `i` of `pBuffers`, `pOffsets` and `pSizes` replace the current state for the transform feedback binding `firstBinding + i`, for `i` in `[0, bindingCount)`. The transform feedback binding is updated to start at the offset indicated by `pOffsets[i]` from the start of the buffer `pBuffers[i]`. 

1446 | Chapter 26. Fixed-Function Vertex Post-Processing
Valid Usage

- VUID-vkCmdBindTransformFeedbackBuffersEXT-transformFeedback-02355
  VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback must be enabled

- VUID-vkCmdBindTransformFeedbackBuffersEXT-firstBinding-02356
  firstBinding must be less than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers

- VUID-vkCmdBindTransformFeedbackBuffersEXT-firstBinding-02357
  The sum of firstBinding and bindingCount must be less than or equal to VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pOffsets-02358
  All elements of pOffsets must be less than the size of the corresponding element in pBuffers

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pOffsets-02359
  All elements of pOffsets must be a multiple of 4

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pBuffers-02360
  All elements of pBuffers must have been created with the VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT flag

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pSize-02361
  If the optional pSize array is specified, each element of pSizes must either be VK_WHOLE_SIZE, or be less than or equal to VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferSize

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pSizes-02362
  All elements of pSizes must be either VK_WHOLE_SIZE, or less than or equal to the size of the corresponding buffer in pBuffers

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pOffsets-02363
  All elements of pOffsets plus pSizes, where the pSizes, element is not VK_WHOLE_SIZE, must be less than or equal to the size of the corresponding buffer in pBuffers

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pBuffers-02364
  Each element of pBuffers that is non-sparse must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdBindTransformFeedbackBuffersEXT-None-02365
  Transform feedback must not be active when the vkCmdBindTransformFeedbackBuffersEXT command is recorded
Valid Usage (Implicit)

- VUID-vkCmdBindTransformFeedbackBuffersEXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pBuffers-parameter
  pBuffers must be a valid pointer to an array of bindingCount valid VkBuffer handles

- VUID-vkCmdBindTransformFeedbackBuffersEXT-pOffsets-parameter
  pOffsets must be a valid pointer to an array of bindingCount VkDeviceSize values

- VUID-vkCmdBindTransformFeedbackBuffersEXT-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdBindTransformFeedbackBuffersEXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdBindTransformFeedbackBuffersEXT-bindingCount-arraylength
  bindingCount must be greater than 0

- VUID-vkCmdBindTransformFeedbackBuffersEXT-commonparent
  Both of commandBuffer, and the elements of pBuffers must have been created, allocated, or retrieved from the same VkDevice

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>

Transform feedback for specific transform feedback buffers is made active by calling:
// Provided by VK_EXT_transform_feedback

void vkCmdBeginTransformFeedbackEXT(
    VkCommandBuffer commandBuffer,              
    uint32_t firstCounterBuffer,               
    uint32_t counterBufferCount,               
    const VkBuffer* pCounterBuffers,           
    const VkDeviceSize* pCounterBufferOffsets);

• commandBuffer is the command buffer into which the command is recorded.

• firstCounterBuffer is the index of the first transform feedback buffer corresponding to
  pCounterBuffers[0] and pCounterBufferOffsets[0].

• counterBufferCount is the size of the pCounterBuffers and pCounterBufferOffsets arrays.

• pCounterBuffers is an optional array of buffer handles to the counter buffers which contain a 4
  byte integer value representing the byte offset from the start of the corresponding transform
  feedback buffer from where to start capturing vertex data. If the byte offset stored to the
  counter buffer location was done using vkCmdEndTransformFeedbackEXT it can be used to
  resume transform feedback from the previous location. If pCounterBuffers is NULL, then
  transform feedback will start capturing vertex data to byte offset zero in all bound transform
  feedback buffers. For each element of pCounterBuffers that is VK_NULL_HANDLE, transform
  feedback will start capturing vertex data to byte zero in the corresponding bound transform
  feedback buffer.

• pCounterBufferOffsets is an optional array of offsets within each of the pCounterBuffers where
  the counter values were previously written. The location in each counter buffer at these offsets
  must be large enough to contain 4 bytes of data. This data is the number of bytes captured by
  the previous transform feedback to this buffer. If pCounterBufferOffsets is NULL, then it is
  assumed the offsets are zero.

The active transform feedback buffers will capture primitives emitted from the corresponding
XfbBuffer in the bound graphics pipeline. Any XfbBuffer emitted that does not output to an active
transform feedback buffer will not be captured.
Valid Usage

- VUID-vkCmdBeginTransformFeedbackEXT-transformFeedback-02366
  
  **VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback must be enabled**

- VUID-vkCmdBeginTransformFeedbackEXT-None-02367
  
  Transform feedback **must not be active**

- VUID-vkCmdBeginTransformFeedbackEXT-firstCounterBuffer-02368
  
  **firstCounterBuffer must be less than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers**

- VUID-vkCmdBeginTransformFeedbackEXT-firstCounterBuffer-02369
  
  The sum of **firstCounterBuffer** and **counterBufferCount** **must be less than or equal to** VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers

- VUID-vkCmdBeginTransformFeedbackEXT-pCounterBuffer-02371
  
  If **pCounterBuffer** is **NULL**, then **pCounterBufferOffsets** **must also be NULL**

- VUID-vkCmdBeginTransformFeedbackEXT-pCounterBuffer-02372
  
  For each buffer handle in the **pCounterBuffers** array that is not **VK_NULL_HANDLE** it **must have been created with a usage value containing VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT**

- VUID-vkCmdBeginTransformFeedbackEXT-None-04128
  
  The last vertex processing stage of the bound graphics pipeline **must have been declared with the Xfb execution mode**

- VUID-vkCmdBeginTransformFeedbackEXT-None-02373
  
  Transform feedback **must not be made active in a render pass instance with multiview enabled**
Valid Usage (Implicit)

- VUID-vkCmdBeginTransformFeedbackEXT-commandBuffer-parameter
  
  **commandBuffer** must be a valid **VkCommandBuffer** handle

- VUID-vkCmdBeginTransformFeedbackEXT-pCounterBufferOffsets-parameter
  
  If **counterBufferCount** is not 0, and **pCounterBufferOffsets** is not NULL, **pCounterBufferOffsets** must be a valid pointer to an array of **counterBufferCount** **VkDeviceSize** values

- VUID-vkCmdBeginTransformFeedbackEXT-commandBuffer-recording
  
  **commandBuffer** must be in the recording state

- VUID-vkCmdBeginTransformFeedbackEXT-commandBuffer-cmdpool
  
  The **VkCommandPool** that **commandBuffer** was allocated from must support graphics operations

- VUID-vkCmdBeginTransformFeedbackEXT-renderpass
  
  This command must only be called inside of a render pass instance

- VUID-vkCmdBeginTransformFeedbackEXT-commonparent
  
  Both of **commandBuffer**, and the elements of **pCounterBuffers** that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same **VkDevice**

Host Synchronization

- Host access to **commandBuffer** must be externally synchronized

- Host access to the **VkCommandPool** that **commandBuffer** was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Transform feedback for specific transform feedback buffers is made inactive by calling:

```c
// Provided by VK_EXT_transform_feedback
void vkCmdEndTransformFeedbackEXT(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstCounterBuffer,
    uint32_t                                    counterBufferCount,
    const VkBuffer*                             pCounterBuffers,
    const VkDeviceSize*                         pCounterBufferOffsets);
```
• commandBuffer is the command buffer into which the command is recorded.
• firstCounterBuffer is the index of the first transform feedback buffer corresponding to pCounterBuffers[0] and pCounterBufferOffsets[0].
• counterBufferCount is the size of the pCounterBuffers and pCounterBufferOffsets arrays.
• pCounterBuffers is an optional array of buffer handles to the counter buffers used to record the current byte positions of each transform feedback buffer where the next vertex output data would be captured. This can be used by a subsequent vkCmdBeginTransformFeedbackEXT call to resume transform feedback capture from this position. It can also be used by vkCmdDrawIndirectByteCountEXT to determine the vertex count of the draw call.
• pCounterBufferOffsets is an optional array of offsets within each of the pCounterBuffers where the counter values can be written. The location in each counter buffer at these offsets must be large enough to contain 4 bytes of data. The data stored at this location is the byte offset from the start of the transform feedback buffer binding where the next vertex data would be written. If pCounterBufferOffsets is NULL, then it is assumed the offsets are zero.

Valid Usage

• VUID-vkCmdEndTransformFeedbackEXT-transformFeedback-02374
  VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback must be enabled
• VUID-vkCmdEndTransformFeedbackEXT-None-02375
  Transform feedback must be active
• VUID-vkCmdEndTransformFeedbackEXT-firstCounterBuffer-02376
  firstCounterBuffer must be less than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers
• VUID-vkCmdEndTransformFeedbackEXT-firstCounterBuffer-02377
  The sum of firstCounterBuffer and counterBufferCount must be less than or equal to VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBuffers
• VUID-vkCmdEndTransformFeedbackEXT-counterBufferCount-02608
  If counterBufferCount is not 0, and pCounterBuffers is not NULL, pCounterBuffers must be a valid pointer to an array of counterBufferCount VkBuffer handles that are either valid or VK_NULL_HANDLE
• VUID-vkCmdEndTransformFeedbackEXT-pCounterBufferOffsets-02378
  For each buffer handle in the array, if it is not VK_NULL_HANDLE it must reference a buffer large enough to hold 4 bytes at the corresponding offset from the pCounterBufferOffsets array
• VUID-vkCmdEndTransformFeedbackEXT-pCounterBuffer-02379
  If pCounterBuffer is NULL, then pCounterBufferOffsets must also be NULL
• VUID-vkCmdEndTransformFeedbackEXT-pCounterBuffers-02380
  For each buffer handle in the pCounterBuffers array that is not VK_NULL_HANDLE it must have been created with a usage value containing VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT
Valid Usage (Implicit)

- **VUID-vkCmdEndTransformFeedbackEXT-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdEndTransformFeedbackEXT-pCounterBufferOffsets-parameter**
  
  If `counterBufferCount` is not 0, and `pCounterBufferOffsets` is not NULL, `pCounterBufferOffsets` must be a valid pointer to an array of `counterBufferCount` `VkDeviceSize` values

- **VUID-vkCmdEndTransformFeedbackEXT-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdEndTransformFeedbackEXT-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- **VUID-vkCmdEndTransformFeedbackEXT-renderpass**
  
  This command must only be called inside of a render pass instance

- **VUID-vkCmdEndTransformFeedbackEXT-commonparent**
  
  Both of `commandBuffer`, and the elements of `pCounterBuffers` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Inside</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>

26.2. Viewport Swizzle

Each primitive sent to a given viewport has a swizzle and optional negation applied to its clip coordinates. The swizzle that is applied depends on the viewport index, and is controlled by the `VkPipelineViewportSwizzleStateCreateInfoNV` pipeline state:
// Provided by VK_NV_viewport_swizzle
typedef struct VkPipelineViewportSwizzleStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkPipelineViewportSwizzleStateCreateFlagsNV flags;
    uint32_t viewportCount;
    const VkViewportSwizzleNV* pViewportSwizzles;
} VkPipelineViewportSwizzleStateCreateInfoNV;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **viewportCount** is the number of viewport swizzles used by the pipeline.
- **pViewportSwizzles** is a pointer to an array of VkViewportSwizzleNV structures, defining the viewport swizzles.

### Valid Usage

- VUID-VkPipelineViewportSwizzleStateCreateInfoNV-viewportCount-01215
  
  `viewportCount` must be greater than or equal to the `viewportCount` set in `VkPipelineViewportStateCreateInfo`.

### Valid Usage (Implicit)

- VUID-VkPipelineViewportSwizzleStateCreateInfoNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SWIZZLE_STATE_CREATE_INFO_NV`.

- VUID-VkPipelineViewportSwizzleStateCreateInfoNV-flags-zerobitmask
  
  `flags` must be 0.

- VUID-VkPipelineViewportSwizzleStateCreateInfoNV-pViewportSwizzles-parameter
  
  `pViewportSwizzles` must be a valid pointer to an array of `viewportCount` valid `VkViewportSwizzleNV` structures.

- VUID-VkPipelineViewportSwizzleStateCreateInfoNV-viewportCount-arraylength
  
  `viewportCount` must be greater than 0.

// Provided by VK_NV_viewport_swizzle
typedef VkFlags VkPipelineViewportSwizzleStateCreateFlagsNV;

VkPipelineViewportSwizzleStateCreateFlagsNV is a bitmask type for setting a mask, but is currently reserved for future use.

The VkPipelineViewportSwizzleStateCreateInfoNV state is set by adding this structure to the `pNext` chain of a VkPipelineViewportStateCreateInfo structure and setting the graphics pipeline state with
vkCreateGraphicsPipelines.

Each viewport specified from 0 to `viewportCount` - 1 has its x, y, z, w swizzle state set to the corresponding x, y, z and w in the `VkViewportSwizzleNV` structure. Each component is of type `VkViewportCoordinateSwizzleNV`, which determines the type of swizzle for that component. The value of x computes the new x component of the position as:

```c
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_X_NV) x' = x;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_X_NV) x' = -x;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Y_NV) x' = y;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Y_NV) x' = -y;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Z_NV) x' = z;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Z_NV) x' = -z;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_W_NV) x' = w;
if (x == VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_W_NV) x' = -w;
```

Similar selections are performed for the y, z, and w coordinates. This swizzling is applied before clipping and perspective divide. If the swizzle for an active viewport index is not specified, the swizzle for x is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_X_NV`, y is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Y_NV`, z is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Z_NV` and w is `VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_W_NV`.

Viewport swizzle parameters are specified by setting the pNext pointer of `VkGraphicsPipelineCreateInfo` to point to a `VkPipelineViewportSwizzleStateCreateInfoNV` structure. `VkPipelineViewportSwizzleStateCreateInfoNV` uses `VkViewportSwizzleNV` to set the viewport swizzle parameters.

The `VkViewportSwizzleNV` structure is defined as:

```c
// Provided by VK_NV_viewport_swizzle
typedef struct VkViewportSwizzleNV {
    VkViewportCoordinateSwizzleNV x;
    VkViewportCoordinateSwizzleNV y;
    VkViewportCoordinateSwizzleNV z;
    VkViewportCoordinateSwizzleNV w;
} VkViewportSwizzleNV;
```

- x is a `VkViewportCoordinateSwizzleNV` value specifying the swizzle operation to apply to the x component of the primitive
- y is a `VkViewportCoordinateSwizzleNV` value specifying the swizzle operation to apply to the y component of the primitive
- z is a `VkViewportCoordinateSwizzleNV` value specifying the swizzle operation to apply to the z component of the primitive
- w is a `VkViewportCoordinateSwizzleNV` value specifying the swizzle operation to apply to the w component of the primitive
Valid Usage (Implicit)

- **VUID-VkViewportSwizzleNV-x-parameter**
  - \textit{x must be a valid} \texttt{VkViewportCoordinateSwizzleNV} \textit{value}

- **VUID-VkViewportSwizzleNV-y-parameter**
  - \textit{y must be a valid} \texttt{VkViewportCoordinateSwizzleNV} \textit{value}

- **VUID-VkViewportSwizzleNV-z-parameter**
  - \textit{z must be a valid} \texttt{VkViewportCoordinateSwizzleNV} \textit{value}

- **VUID-VkViewportSwizzleNV-w-parameter**
  - \textit{w must be a valid} \texttt{VkViewportCoordinateSwizzleNV} \textit{value}

Possible values of the \texttt{VkViewportSwizzleNV::x, y, z,} and \textit{w} members, specifying swizzling of the corresponding components of primitives, are:

```c
// Provided by VK_NV_viewport_swizzle
typedef enum VkViewportCoordinateSwizzleNV {
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_X_NV = 0,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_X_NV = 1,
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Y_NV = 2,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Y_NV = 3,
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_Z_NV = 4,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_Z_NV = 5,
    VK_VIEWPORT_COORDINATE_SWIZZLE_POSITIVE_W_NV = 6,
    VK_VIEWPORT_COORDINATE_SWIZZLE_NEGATIVE_W_NV = 7,
} VkViewportCoordinateSwizzleNV;
```

These values are described in detail in \textit{Viewport Swizzle}.

### 26.3. Flat Shading

\textit{Flat shading} a vertex output attribute means to assign all vertices of the primitive the same value for that output. The output values assigned are those of the \textit{provoking vertex} of the primitive. Flat shading is applied to those vertex attributes that \textit{match} fragment input attributes which are decorated as \texttt{Flat}.

If neither \texttt{geometry} nor \texttt{tessellation shading} is active, the provoking vertex is determined by the \texttt{primitive topology} defined by \texttt{VkPipelineInputAssemblyStateCreateInfo::topology} used to execute the \textit{drawing command}.

If \texttt{geometry shading} is active, the provoking vertex is determined by the \texttt{primitive topology} defined by the \texttt{OutputPoints, OutputLineStrips,} or \texttt{OutputTriangleStrips} execution mode.

If \texttt{tessellation shading} is active but \texttt{geometry shading} is not, the provoking vertex \textbf{may} be any of the vertices in each primitive.

For a given primitive topology, the pipeline’s provoking vertex mode determines which vertex is
the provoking vertex. To specify the provoking vertex mode, attach an instance of
\texttt{VkPipelineRasterizationProvokingVertexStateCreateInfoEXT} to 
\texttt{VkPipelineRasterizationStateCreateInfo::pNext} when creating the pipeline.

The \texttt{VkPipelineRasterizationProvokingVertexStateCreateInfoEXT} structure is defined as:

\begin{verbatim}
// Provided by VK_EXT_provoking_vertex
typedef struct VkPipelineRasterizationProvokingVertexStateCreateInfoEXT {
    VkStructureType             sType;
    const void*                 pNext;
    VkProvokingVertexModeEXT    provokingVertexMode;
} VkPipelineRasterizationProvokingVertexStateCreateInfoEXT;
\end{verbatim}

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to an extension-specific structure.
- \texttt{provokingVertexMode} is a \texttt{VkProvokingVertexModeEXT} value selecting the provoking vertex mode.

If this struct is not provided when creating the pipeline, the pipeline will use the
\texttt{VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT} mode.

If the \texttt{provokingVertexModePerPipeline} limit is \texttt{VK_FALSE}, then the all pipelines bound within a
renderpass instance \textbf{must} have the same \texttt{provokingVertexMode}.

**Valid Usage**

- VUID-VkPipelineRasterizationProvokingVertexStateCreateInfoEXT-provokingVertexMode-04883
  If \texttt{provokingVertexMode} is \texttt{VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT}, then the
  \texttt{provokingVertexLast} feature \textbf{must} be enabled

**Valid Usage (Implicit)**

- VUID-VkPipelineRasterizationProvokingVertexStateCreateInfoEXT-sType-sType
  \texttt{sType} \textbf{must} be \texttt{VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_PROVOKING_VERTEX_STATE_CREATE_INFO_EXT}
- VUID-VkPipelineRasterizationProvokingVertexStateCreateInfoEXT-provokingVertexMode-parameter
  \texttt{provokingVertexMode} \textbf{must} be a valid \texttt{VkProvokingVertexModeEXT} value

Possible values of \texttt{VkPipelineRasterizationProvokingVertexStateCreateInfoEXT::provokingVertexMode} are:
// Provided by VK_EXT_provoking_vertex
typedef enum VkProvokingVertexModeEXT {
    VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT = 0,
    VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT = 1,
} VkProvokingVertexModeEXT;

- **VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT** specifies that the provoking vertex is the first non-adjacency vertex in the list of vertices used by a primitive.

- **VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT** specifies that the provoking vertex is the last non-adjacency vertex in the list of vertices used by a primitive.

These modes are described more precisely in Primitive Topologies.

### 26.4. Primitive Clipping

Primitives are culled against the *cull volume* and then clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by:

\[
-w_c \leq x_c \leq w_c \\
-w_c \leq y_c \leq w_c \\
0 \leq z_c \leq w_c
\]

This view volume can be further restricted by as many as VkPhysicalDeviceLimits::maxClipDistances client-defined half-spaces.

The cull volume is the intersection of up to VkPhysicalDeviceLimits::maxCullDistances client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped).

A shader must write a single cull distance for each enabled cull half-space to elements of the CullDistance array. If the cull distance for any enabled cull half-space is negative for all of the vertices of the primitive under consideration, the primitive is discarded. Otherwise the primitive is clipped against the clip volume as defined below.

The clip volume is the intersection of up to VkPhysicalDeviceLimits::maxClipDistances client-defined half-spaces with the view volume (if no client-defined clip half-spaces are enabled, the clip volume is the view volume).

A shader must write a single clip distance for each enabled clip half-space to elements of the ClipDistance array. Clip half-space i is then given by the set of points satisfying the inequality

\[c_i(P) \geq 0\]

where \(c_i(P)\) is the clip distance i at point \(P\). For point primitives, \(c_i(P)\) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections Basic Line Segment Rasterization and Basic Polygon Rasterization, using the perspective interpolation equations.
The number of client-defined clip and cull half-spaces that are enabled is determined by the explicit size of the built-in arrays `ClipDistance` and `CullDistance`, respectively, declared as an output in the interface of the entry point of the final shader stage before clipping.

If `VkPipelineRasterizationDepthClipStateCreateInfoEXT` is present in the graphics pipeline state then depth clipping is disabled if ` VkPipelineRasterizationDepthClipStateCreateInfoEXT::depthClipEnable` is `VK_FALSE`. Otherwise, if `VkPipelineRasterizationDepthClipStateCreateInfoEXT` is not present, depth clipping is disabled when `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is `VK_TRUE`. When depth clipping is disabled, the plane equation

\[ 0 \leq z_c \leq w_c \]

(see the clip volume definition above) is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point or line segment, then clipping passes it unchanged if its vertices lie entirely within the clip volume.

Possible values of `VkPhysicalDevicePointClippingProperties::pointClippingBehavior`, specifying clipping behavior of a point primitive whose vertex lies outside the clip volume, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkPointClippingBehavior {
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES = 0,
    VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY = 1,
    // Provided by VK_KHR_maintenance2
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES_KHR =
    VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES,
    // Provided by VK_KHR_maintenance2
    VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY_KHR =
    VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY,
} VkPointClippingBehavior;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance2
typedef VkPointClippingBehavior VkPointClippingBehaviorKHR;
```

- `VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES` specifies that the primitive is discarded if the vertex lies outside any clip plane, including the planes bounding the view volume.
- `VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY` specifies that the primitive is discarded only if the vertex lies outside any user clip plane.

If either of a line segment's vertices lie outside of the clip volume, the line segment may be clipped, with new vertex coordinates computed for each vertex that lies outside the clip volume. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.
This clipping produces a value, \(0 \leq t \leq 1\), for each clipped vertex. If the coordinates of a clipped vertex are \(P\) and the unclipped line segment's vertex coordinates are \(P_1\) and \(P_2\), then \(t\) satisfies the following equation

\[ P = tP_1 + (1-t)P_2. \]

\(t\) is used to clip vertex output attributes as described in Clipping Shader Outputs.

If the primitive is a polygon, it passes unchanged if every one of its edges lies entirely inside the clip volume, and is either clipped or discarded otherwise. If the edges of the polygon intersect the boundary of the clip volume, the intersecting edges are reconnected by new edges that lie along the boundary of the clip volume - in some cases requiring the introduction of new vertices into a polygon.

If a polygon intersects an edge of the clip volume's boundary, the clipped polygon must include a point on this boundary edge.

Primitives rendered with user-defined half-spaces must satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex \(i\) has a single specified clip distance \(d_i\) (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by \(-d_i\) (and the graphics pipeline is otherwise the same). In this case, primitives must not be missing any pixels, and pixels must not be drawn twice in regions where those primitives are cut by the clip planes.

### 26.5. Clipping Shader Outputs

Next, vertex output attributes are clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Let the output values assigned to the two vertices \(P_1\) and \(P_2\) of an unclipped edge be \(c_1\) and \(c_2\). The value of \(t\) (see Primitive Clipping) for a clipped point \(P\) is used to obtain the output value associated with \(P\) as

\[ c = t c_1 + (1-t) c_2. \]

(Multiplying an output value by a scalar means multiplying each of \(x, y, z,\) and \(w\) by the scalar.)

Since this computation is performed in clip space before division by \(w_c\), clipped output values are perspective-correct.

Polygon clipping creates a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Output value clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

For vertex output attributes whose matching fragment input attributes are decorated with NoPerspective, the value of \(t\) used to obtain the output value associated with \(P\) will be adjusted to
produce results that vary linearly in framebuffer space.

Output attributes of integer or unsigned integer type must always be flat shaded. Flat shaded attributes are constant over the primitive being rasterized (see Basic Line Segment Rasterization and Basic Polygon Rasterization), and no interpolation is performed. The output value $c$ is taken from either $c_1$ or $c_2$, since flat shading has already occurred and the two values are identical.

### 26.6. Controlling Viewport W Scaling

If viewport W scaling is enabled, the W component of the clip coordinate is modified by the provided coefficients from the corresponding viewport as follows.

$$w'_c = x_{coeff} x_c + y_{coeff} y_c + w_c$$

The *VkPipelineViewportWScalingStateCreateInfoNV* structure is defined as:

```c
// Provided by VK_NV_clip_space_w_scaling
typedef struct VkPipelineViewportWScalingStateCreateInfoNV {
    VkStructureType                sType;
    const void*                    pNext;
    VkBool32                       viewportWScalingEnable;
    uint32_t                       viewportCount;
    const VkViewportWScalingNV*    pViewportWScalings;
} VkPipelineViewportWScalingStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `viewportWScalingEnable` controls whether viewport W scaling is enabled.
- `viewportCount` is the number of viewports used by W scaling, and must match the number of viewports in the pipeline if viewport W scaling is enabled.
- `pViewportWScalings` is a pointer to an array of *VkViewportWScalingNV* structures defining the W scaling parameters for the corresponding viewports. If the viewport W scaling state is dynamic, this member is ignored.

**Valid Usage (Implicit)**

- `VUID-VkPipelineViewportWScalingStateCreateInfoNV-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV`

- `VUID-VkPipelineViewportWScalingStateCreateInfoNV-viewportCount-arraylength`  
  `viewportCount` must be greater than 0

The *VkPipelineViewportWScalingStateCreateInfoNV* state is set by adding this structure to the `pNext` chain of a *VkPipelineViewportStateCreateInfo* structure and setting the graphics pipeline state with `vkCreateGraphicsPipelines`.
If the bound pipeline state object was not created with the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled, viewport W scaling parameters are specified using the `pViewportWScalings` member of `VkPipelineViewportWScalingStateCreateInfoNV` in the pipeline state object. If the pipeline state object was created with the `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic state enabled, the viewport transformation parameters are dynamically set and changed with the command:

```c
// Provided by VK_NV_clip_space_w_scaling
void vkCmdSetViewportWScalingNV(
    VkCommandBuffer                             commandBuffer,   // Provided by VK_NV_clip_space_w_scaling
    uint32_t                                    firstViewport,     // Provided by VK_NV_clip_space_w_scaling
    uint32_t                                    viewportCount,     // Provided by VK_NV_clip_space_w_scaling
    const VkViewportWScalingNV*                 pViewportWScalings);   // Provided by VK_NV_clip_space_w_scaling
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstViewport` is the index of the first viewport whose parameters are updated by the command.
- `viewportCount` is the number of viewports whose parameters are updated by the command.
- `pViewportWScalings` is a pointer to an array of `VkViewportWScalingNV` structures specifying viewport parameters.

The viewport parameters taken from element i of `pViewportWScalings` replace the current state for the viewport index `firstViewport + i`, for i in `[0, viewportCount)`.  

**Valid Usage**

- `VUID-vkCmdSetViewportWScalingNV-firstViewport-01324`

  The sum of `firstViewport` and `viewportCount` must be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive.
Valid Usage (Implicit)

- VUID-vkCmdSetViewportWScalingNV-commandBuffer-parameter
  \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle

- VUID-vkCmdSetViewportWScalingNV-pViewportWScalings-parameter
  \texttt{pViewportWScalings} must be a valid pointer to an array of \texttt{viewportCount} \texttt{VkViewportWScalingNV} structures

- VUID-vkCmdSetViewportWScalingNV-commandBuffer-recording
  \texttt{commandBuffer} must be in the recording state

- VUID-vkCmdSetViewportWScalingNV-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics operations

- VUID-vkCmdSetViewportWScalingNV-viewportCount-arraylength
  \texttt{viewportCount} must be greater than 0

Host Synchronization

- Host access to \texttt{commandBuffer} must be externally synchronized

- Host access to the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both \texttt{VkPipelineViewportWScalingStateCreateInfoNV} and \texttt{vkCmdSetViewportWScalingNV} use \texttt{VkViewportWScalingNV} to set the viewport transformation parameters.

The \texttt{VkViewportWScalingNV} structure is defined as:

```c
// Provided by VK_NV_clip_space_w_scaling
typedef struct VkViewportWScalingNV {
  float xcoeff;
  float ycoeff;
} VkViewportWScalingNV;
```

- \texttt{xcoeff} and \texttt{ycoeff} are the viewport’s W scaling factor for x and y respectively.
26.7. Coordinate Transformations

Clip coordinates for a vertex result from shader execution, which yields a vertex coordinate Position.

Perspective division on clip coordinates yields normalized device coordinates, followed by a viewport transformation (see Controlling the Viewport) to convert these coordinates into framebuffer coordinates.

If a vertex in clip coordinates has a position given by

\[
\begin{pmatrix}
  x_c \\
  y_c \\
  z_c \\
  w_c
\end{pmatrix}
\]

then the vertex's normalized device coordinates are

\[
\begin{pmatrix}
  x_d \\
  y_d \\
  z_d
\end{pmatrix} = \begin{pmatrix}
  x_c \\
  w_c \\
  y_c \\
  w_c \\
  z_c \\
  w_c
\end{pmatrix}
\]

26.8. Render Pass Transform

A render pass transform can be enabled for render pass instances. The clip coordinates \((x_c, y_c)\) that result from vertex shader execution are transformed by a rotation of 0, 90, 180, or 270 degrees in the XY plane, centered at the origin.

When Render pass transform is enabled, the transform applies to all primitives for all subpasses of the render pass. The transformed vertex in clip coordinates has a position given by

\[
\begin{pmatrix}
  x_{c\text{trans}} \\
  y_{c\text{trans}} \\
  z_{c\text{trans}}
\end{pmatrix} = \begin{pmatrix}
  x_c \cos \theta - y_c \sin \theta \\
  x_c \sin \theta + y_c \cos \theta \\
  z_c
\end{pmatrix}
\]

where

- \(\theta\) is 0 degrees for VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR
- \(\theta\) is 90 degrees for VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR
- \(\theta\) is 180 degrees for VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR
- \(\theta\) is 270 degrees for VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR

The transformed vertex's normalized device coordinates are

\[
\begin{pmatrix}
  x_d \\
  y_d \\
  z_d
\end{pmatrix} = \begin{pmatrix}
  x_{c\text{trans}}/w_c \\
  y_{c\text{trans}}/w_c \\
  z_{c\text{trans}}/w_c
\end{pmatrix}
\]
When render pass transform is enabled for a renderpass instance, the following additional features are enabled:

- Each `VkViewport` specified by either `VkPipelineViewportStateCreateInfo::pViewports` or `vkCmdSetViewport` will have its width/height \((p_x, p_y)\) and its center \((o_x, o_y)\) similarly transformed by the implementation.

- Each scissor specified by `VkPipelineViewportStateCreateInfo::pScissors` or `vkCmdSetScissor` will have its \((\text{offset}_x, \text{offset}_y)\) and \((\text{extent}_x, \text{extent}_y)\) similarly transformed by the implementation.

- The `renderArea` specified in `VkCommandBufferInheritanceRenderPassTransformInfoQCOM` and `VkRenderPassBeginInfo` will be similarly transformed by the implementation.

- The \((x, y)\) components of shader variables with built-in decorations `FragCoord`, `SamplePosition`, or `PointCoord` will be similarly transformed by the implementation.

- The \((x,y)\) components of the `offset` operand of the `InterpolateAtOffset` extended instruction will be similarly transformed by the implementation.

- The values returned by SPIR-V derivative instructions `OpDPdx`, `OpDPdy`, `OpDPdxCourse`, `OpDPdyCourse`, `OpDPdxFine`, `OpDPdyFine` will be similarly transformed by the implementation.

The net result of the above, is that applications can act as if rendering to a framebuffer oriented with the `VkSurfaceCapabilitiesKHR::currentTransform`. In other words, applications can act as if the presentation engine will be performing the transformation of the swapchain image after rendering and prior to presentation to the user. In fact, the transformation of the various items cited above are being handled by the implementation as the rendering takes place.

### 26.9. Controlling the Viewport

The viewport transformation is determined by the selected viewport's width and height in pixels, \(p_x\) and \(p_y\), respectively, and its center \((o_x, o_y)\) (also in pixels), as well as its depth range min and max determining a depth range scale value \(p_z\) and a depth range bias value \(o_z\) (defined below). The vertex's framebuffer coordinates \((x_f, y_f, z_f)\) are given by

\[
x_f = \left(\frac{p_x}{2}\right) x_d + o_x
\]

\[
y_f = \left(\frac{p_y}{2}\right) y_d + o_y
\]

\[
z_f = p_z \times z_d + o_z
\]

Multiple viewports are available, numbered zero up to `VkPhysicalDeviceLimits::maxViewports` minus one. The number of viewports used by a pipeline is controlled by the `viewportCount` member of the `VkPipelineViewportStateCreateInfo` structure used in pipeline creation.

The `VkPipelineViewportStateCreateInfo` structure is defined as:
\[
\text{typedef struct VkPipelineViewportStateCreateInfo} \{
    \text{VkStructureType} \quad \text{sType};
    \text{const void*} \quad \text{pNext};
    \text{VkPipelineViewportStateCreateFlags} \quad \text{flags};
    \text{uint32_t} \quad \text{viewportCount};
    \text{const VkViewport*} \quad \text{pViewports};
    \text{uint32_t} \quad \text{scissorCount};
    \text{const VkRect2D*} \quad \text{pScissors};
\} \text{VkPipelineViewportStateCreateInfo};
\]

- \text{sType} is the type of this structure.
- \text{pNext} is \text{NULL} or a pointer to a structure extending this structure.
- \text{flags} is reserved for future use.
- \text{viewportCount} is the number of viewports used by the pipeline.
- \text{pViewports} is a pointer to an array of \text{VkViewport} structures, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
- \text{scissorCount} is the number of \text{scissors} and \text{must} match the number of viewports.
- \text{pScissors} is a pointer to an array of \text{VkRect2D} structures defining the rectangular bounds of the scissor for the corresponding viewport. If the scissor state is dynamic, this member is ignored.
Valid Usage

- **VUID-VkPipelineViewportStateCreateInfo-viewportCount-01216**
  If the **multiple viewports** feature is not enabled, **viewportCount must not be greater than 1**

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-01217**
  If the **multiple viewports** feature is not enabled, **scissorCount must not be greater than 1**

- **VUID-VkPipelineViewportStateCreateInfo-viewportCount-01218**
  **viewportCount must be less than or equal to** `VkPhysicalDeviceLimits::maxViewports`

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-01219**
  **scissorCount must be less than or equal to** `VkPhysicalDeviceLimits::maxViewports`

- **VUID-VkPipelineViewportStateCreateInfo-x-02821**
  The x and y members of **offset** member of any element of **pScissors must be greater than or equal to 0**

- **VUID-VkPipelineViewportStateCreateInfo-offset-02822**
  Evaluation of \((offset.x + extent.width)\) must not cause a signed integer addition overflow for any element of **pScissors**

- **VUID-VkPipelineViewportStateCreateInfo-offset-02823**
  Evaluation of \((offset.y + extent.height)\) must not cause a signed integer addition overflow for any element of **pScissors**

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-04134**
  **If the graphics pipeline is being created without** `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` **and** `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` **set then scissorCount and viewportCount must be identical**

- **VUID-VkPipelineViewportStateCreateInfo-viewportCount-04135**
  **If the graphics pipeline is being created with** `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` **set then viewportCount must be 0, otherwise it must be greater than 0**

- **VUID-VkPipelineViewportStateCreateInfo-scissorCount-04136**
  **If the graphics pipeline is being created with** `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` **set then scissorCount must be 0, otherwise it must be greater than 0**

- **VUID-VkPipelineViewportStateCreateInfo-viewportWScalingEnable-01726**
  **If the viewportWScalingEnable member of a VkPipelineViewportWScalingStateCreateInfoNV structure included in the pNext chain is VK_TRUE, the viewportCount member of the VkPipelineViewportWScalingStateCreateInfoNV structure must be greater than or equal to** `VkPipelineViewportStateCreateInfo::viewportCount`
Valid Usage (Implicit)

- **VUID-VkPipelineViewportStateCreateInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO`

- **VUID-VkPipelineViewportStateCreateInfo-pNext-pNext**
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of:
    - `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV`
    - `VkPipelineViewportExclusiveScissorStateCreateInfoNV`
    - `VkPipelineViewportShadingRateImageStateCreateInfoNV`
    - `VkPipelineViewportSwizzleStateCreateInfoNV`
    - `VkPipelineViewportWScalingStateCreateInfoNV`

- **VUID-VkPipelineViewportStateCreateInfo-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique

- **VUID-VkPipelineViewportStateCreateInfo-flags-zerobitmask**
  - `flags` must be `0`

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled then the viewport count and viewport transformation parameters are set dynamically by calling:

```cpp
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetViewportWithCountEXT(
    VkCommandBuffer commandBuffer,                  // commandBuffer is the command buffer into which the command will be recorded.
    uint32_t viewportCount,                         // viewportCount specifies the viewport count.
    const VkViewport* pViewports);                // pViewports specifies the viewports to use for drawing.
```

Valid Usage

- **VUID-vkCmdSetViewportWithCountEXT-None-03393**
  - The extendedDynamicState feature must be enabled

- **VUID-vkCmdSetViewportWithCountEXT-viewportCount-03394**
  - `viewportCount` must be between `1` and `VkPhysicalDeviceLimits::maxViewports`, inclusive

- **VUID-vkCmdSetViewportWithCountEXT-viewportCount-03395**
  - If the multiple viewports feature is not enabled, `viewportCount` must be `1`

- **VUID-vkCmdSetViewportWithCountEXT-commandBuffer-04819**
  - `commandBuffer` must not have `VkCommandBufferInheritanceViewportScissorInfoNV ::viewportScissor2D` enabled.
Valid Usage (Implicit)

- VUID-vkCmdSetViewportWithCountEXT-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetViewportWithCountEXT-pViewports-parameter
  
  `pViewports` must be a valid pointer to an array of `viewportCount` valid `VkViewport` structures

- VUID-vkCmdSetViewportWithCountEXT-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetViewportWithCountEXT-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdSetViewportWithCountEXT-viewportCount-arraylength
  
  `viewportCount` must be greater than 0

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT` dynamic state enabled then the scissor count and scissor rectangular bounds are set dynamically by calling:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetScissorWithCountEXT(
    VkCommandBuffer commandBuffer,
    uint32_t scissorCount,
    const VkRect2D* pScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `scissorCount` specifies the scissor count.
• `pScissors` specifies the scissors to use for drawing.

### Valid Usage

- **VUID-vkCmdSetScissorWithCountEXT-None-03396**
  
  The `extendedDynamicState` feature **must** be enabled

- **VUID-vkCmdSetScissorWithCountEXT-scissorCount-03397**
  
  `scissorCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive

- **VUID-vkCmdSetScissorWithCountEXT-scissorCount-03398**
  
  If the `multiple viewports` feature is not enabled, `scissorCount` **must** be 1

- **VUID-vkCmdSetScissorWithCountEXT-x-03399**
  
  The `x` and `y` members of `offset` member of any element of `pScissors` **must** be greater than or equal to 0

- **VUID-vkCmdSetScissorWithCountEXT-offset-03400**
  
  Evaluation of `(offset.x + extent.width)` **must** not cause a signed integer addition overflow for any element of `pScissors`

- **VUID-vkCmdSetScissorWithCountEXT-offset-03401**
  
  Evaluation of `(offset.y + extent.height)` **must** not cause a signed integer addition overflow for any element of `pScissors`

- **VUID-vkCmdSetScissorWithCountEXT-commandBuffer-04820**
  
  `commandBuffer` **must** not have `VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D` enabled.

### Valid Usage (Implicit)

- **VUID-vkCmdSetScissorWithCountEXT-commandBuffer-parameter**
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetScissorWithCountEXT-pScissors-parameter**
  
  `pScissors` **must** be a valid pointer to an array of `scissorCount` `VkRect2D` structures

- **VUID-vkCmdSetScissorWithCountEXT-commandBuffer-recording**
  
  `commandBuffer` **must** be in the `recording` state

- **VUID-vkCmdSetScissorWithCountEXT-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

- **VUID-vkCmdSetScissorWithCountEXT-scissorCount-arraylength**
  
  `scissorCount` **must** be greater than 0
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineViewportStateCreateFlags;

`VkPipelineViewportStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

A vertex processing stage can direct each primitive to zero or more viewports. The destination viewports for a primitive are selected by the last active vertex processing stage that has an output variable decorated with `ViewportIndex` (selecting a single viewport) or `ViewportMaskNV` (selecting multiple viewports). The viewport transform uses the viewport corresponding to either the value assigned to `ViewportIndex` or one of the bits set in `ViewportMaskNV`, and taken from an implementation-dependent vertex of each primitive. If `ViewportIndex` or any of the bits in `ViewportMaskNV` are outside the range zero to `viewportCount` minus one for a primitive, or if the last active vertex processing stage did not assign a value to either `ViewportIndex` or `ViewportMaskNV` for all vertices of a primitive due to flow control, the values resulting from the viewport transformation of the vertices of such primitives are undefined. If the last vertex processing stage does not have an output decorated with `ViewportIndex` or `ViewportMaskNV`, the viewport numbered zero is used by the viewport transformation.

A single vertex can be used in more than one individual primitive, in primitives such as `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`. In this case, the viewport transformation is applied separately for each primitive.

If the bound pipeline state object was not created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled, viewport transformation parameters are specified using the `pViewports` member of `VkPipelineViewportStateCreateInfo` in the pipeline state object. If the pipeline state object was created with the `VK_DYNAMIC_STATE_VIEWPORT` dynamic state enabled, the viewport transformation parameters are dynamically set and changed with the command:
// Provided by VK_VERSION_1_0

void vkCmdSetViewport(
    VkCommandBuffer                            commandBuffer, 
    uint32_t                                    firstViewport, 
    uint32_t                                    viewportCount, 
    const VkViewport*                           pViewports);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstViewport` is the index of the first viewport whose parameters are updated by the command.
- `viewportCount` is the number of viewports whose parameters are updated by the command.
- `pViewports` is a pointer to an array of `VkViewport` structures specifying viewport parameters.

The viewport parameters taken from element `i` of `pViewports` replace the current state for the viewport index `firstViewport + i`, for `i` in `[0, `viewportCount`].

---

**Valid Usage**

- VUID-vkCmdSetViewport-firstViewport-01223
  The sum of `firstViewport` and `viewportCount` must be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive

- VUID-vkCmdSetViewport-firstViewport-01224
  If the multiple viewports feature is not enabled, `firstViewport` must be 0

- VUID-vkCmdSetViewport-viewportCount-01225
  If the multiple viewports feature is not enabled, `viewportCount` must be 1

- VUID-vkCmdSetViewport-commandBuffer-04821
  `commandBuffer` must not have `VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D` enabled.
Valid Usage (Implicit)

- VUID-vkCmdSetViewport-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetViewport-pViewports-parameter
  `pViewports` must be a valid pointer to an array of `viewportCount` valid `VkViewport` structures

- VUID-vkCmdSetViewport-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetViewport-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdSetViewport-viewportCount-arraylength
  `viewportCount` must be greater than 0

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both `VkPipelineViewportStateCreateInfo` and `vkCmdSetViewport` use `VkViewport` to set the viewport transformation parameters.

The `VkViewport` structure is defined as:
```c
typedef struct VkViewport {
    float x;
    float y;
    float width;
    float height;
    float minDepth;
    float maxDepth;
} VkViewport;
```

- \(x\) and \(y\) are the viewport's upper left corner \((x, y)\).
- \(width\) and \(height\) are the viewport's width and height, respectively.
- \(minDepth\) and \(maxDepth\) are the depth range for the viewport.

**Note**

Despite their names, \(minDepth\) can be less than, equal to, or greater than \(maxDepth\).

The framebuffer depth coordinate \(z\) may be represented using either a fixed-point or floating-point representation. However, a floating-point representation must be used if the depth/stencil attachment has a floating-point depth component. If an \(m\)-bit fixed-point representation is used, we assume that it represents each value \(k / 2^m - 1\), where \(k \in \{0, 1, \ldots, 2^m - 1\}\), as \(k\) (e.g. 1.0 is represented in binary as a string of all ones).

The viewport parameters shown in the above equations are found from these values as:

\[
o_x = x + \frac{width}{2}
\]

\[
o_y = y + \frac{height}{2}
\]

\[
o_z = minDepth
\]

\[
p_x = width
\]

\[
p_y = height
\]

\[
p_z = maxDepth - minDepth.
\]

If a render pass transform is enabled, the values \((p_x, p_y)\) and \((o_x, o_y)\) defining the viewport are transformed as described in render pass transform before participating in the viewport transform.

The application can specify a negative term for \(height\), which has the effect of negating the \(y\)
coordinate in clip space before performing the transform. When using a negative `height`, the application **should** also adjust the `y` value to point to the lower left corner of the viewport instead of the upper left corner. Using the negative `height` allows the application to avoid having to negate the `y` component of the `Position` output from the last vertex processing stage in shaders that also target other graphics APIs.

The width and height of the **implementation-dependent maximum viewport dimensions** **must** be greater than or equal to the width and height of the largest image which **can** be created and attached to a framebuffer.

The floating-point viewport bounds are represented with an **implementation-dependent precision**.

---

**Valid Usage**

- **VUID-VkViewport-width-01770**
  
  `width` **must** be greater than 0.0

- **VUID-VkViewport-width-01771**
  
  `width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions[0]`

- **VUID-VkViewport-height-01773**
  
  The absolute value of `height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxViewportDimensions[1]`

- **VUID-VkViewport-x-01774**
  
  `x` **must** be greater than or equal to `viewportBoundsRange[0]`

- **VUID-VkViewport-x-01232**
  
  `(x + width)` **must** be less than or equal to `viewportBoundsRange[1]`

- **VUID-VkViewport-y-01775**
  
  `y` **must** be greater than or equal to `viewportBoundsRange[0]`

- **VUID-VkViewport-y-01776**
  
  `y` **must** be less than or equal to `viewportBoundsRange[1]`

- **VUID-VkViewport-y-01777**
  
  `(y + height)` **must** be greater than or equal to `viewportBoundsRange[0]`

- **VUID-VkViewport-y-01233**
  
  `(y + height)` **must** be less than or equal to `viewportBoundsRange[1]`

- **VUID-VkViewport-minDepth-01234**
  
  Unless `VK_EXT_depth_range_unrestricted` extension is enabled `minDepth` **must** be between 0.0 and 1.0, inclusive

- **VUID-VkViewport-maxDepth-01235**
  
  Unless `VK_EXT_depth_range_unrestricted` extension is enabled `maxDepth` **must** be between 0.0 and 1.0, inclusive
Chapter 27. Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each discrete location of this image contains associated data such as depth, color, or other attributes.

Rasterizing a primitive begins by determining which squares of an integer grid in framebuffer coordinates are occupied by the primitive, and assigning one or more depth values to each such square. This process is described below for points, lines, and polygons.

A grid square, including its (x,y) framebuffer coordinates, z (depth), and associated data added by fragment shaders, is called a fragment. A fragment is located by its upper left corner, which lies on integer grid coordinates.

Rasterization operations also refer to a fragment’s sample locations, which are offset by fractional values from its upper left corner. The rasterization rules for points, lines, and triangles involve testing whether each sample location is inside the primitive. Fragments need not actually be square, and rasterization rules are not affected by the aspect ratio of fragments. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other.

We assume that fragments are square, since it simplifies antialiasing and texturing. After rasterization, fragments are processed by fragment operations.

Several factors affect rasterization, including the members of `VkPipelineRasterizationStateCreateInfo` and `VkPipelineMultisampleStateCreateInfo`.

The `VkPipelineRasterizationStateCreateInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationStateCreateFlags flags;
    VkBool32 depthClampEnable;
    VkBool32 rasterizerDiscardEnable;
    VkPolygonMode polygonMode;
    VkCullModeFlags cullMode;
    VkFrontFace frontFace;
    float depthBiasConstantFactor;
    float depthBiasClamp;
    float depthBiasSlopeFactor;
    float lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
• `depthClampEnable` controls whether to clamp the fragment's depth values as described in `Depth Test`. If the pipeline is not created with `VkPipelineRasterizationDepthClipStateCreateInfoEXT` present then enabling depth clamp will also disable clipping primitives to the z planes of the frustrum as described in `Primitive Clipping`. Otherwise depth clipping is controlled by the state set in `VkPipelineRasterizationDepthClipStateCreateInfoEXT`.

• `rasterizerDiscardEnable` controls whether primitives are discarded immediately before the rasterization stage.

• `polygonMode` is the triangle rendering mode. See `VkPolygonMode`.

• `cullMode` is the triangle facing direction used for primitive culling. See `VkCullModeFlagBits`.

• `frontFace` is a `VkFrontFace` value specifying the front-facing triangle orientation to be used for culling.

• `depthBiasEnable` controls whether to bias fragment depth values.

• `depthBiasConstantFactor` is a scalar factor controlling the constant depth value added to each fragment.

• `depthBiasClamp` is the maximum (or minimum) depth bias of a fragment.

• `depthBiasSlopeFactor` is a scalar factor applied to a fragment's slope in depth bias calculations.

• `lineWidth` is the width of rasterized line segments.

The application can also add a `VkPipelineRasterizationStateRasterizationOrderAMD` structure to the `pNext` chain of a `VkPipelineRasterizationStateCreateInfo` structure. This structure enables selecting the rasterization order to use when rendering with the corresponding graphics pipeline as described in `Rasterization Order`.

---

**Valid Usage**

- **VUID-VkPipelineRasterizationStateCreateInfo-depthClampEnable-00782**
  
  If the `depth clamping` feature is not enabled, `depthClampEnable` must be `VK_FALSE`.

- **VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-01507**
  
  If the `non-solid fill modes` feature is not enabled, `polygonMode` must be `VK_POLYGON_MODE_FILL` or `VK_POLYGON_MODE_FILL_RECTANGLE_NV`.

- **VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-01414**
  
  If the `VK_NV_fill_rectangle` extension is not enabled, `polygonMode` must not be `VK_POLYGON_MODE_FILL_RECTANGLE_NV`.

- **VUID-VkPipelineRasterizationStateCreateInfo-pointPolygons-04458**
  
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPhysicalDevicePortabilitySubsetFeaturesKHR::pointPolygons` is `VK_FALSE`, and `rasterizerDiscardEnable` is `VK_FALSE`, `polygonMode` must not be `VK_POLYGON_MODE_POINT`.  

---

Chapter 27. Rasterization | 1477
Valid Usage (Implicit)

- **VUID-VkPipelineRasterizationStateCreateInfo-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO`

- **VUID-VkPipelineRasterizationStateCreateInfo-pNext-pNext**
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain **must** be either `NULL` or a pointer to a valid instance of

  - `VkPipelineRasterizationConservativeStateCreateInfoEXT`,
  - `VkPipelineRasterizationDepthClipStateCreateInfoEXT`,
  - `VkPipelineRasterizationLineStateCreateInfoEXT`,
  - `VkPipelineRasterizationProvokingVertexStateCreateInfoEXT`,
  - `VkPipelineRasterizationStateRasterizationOrderAMD`,
  - `VkPipelineRasterizationStateStreamCreateInfoEXT` or

- **VUID-VkPipelineRasterizationStateCreateInfo-sType-unique**
  
  The `sType` value of each struct in the `pNext` chain **must** be unique

- **VUID-VkPipelineRasterizationStateCreateInfo-flags-zero bitmask**
  
  `flags` **must** be `0`

- **VUID-VkPipelineRasterizationStateCreateInfo-polygonMode-parameter**
  
  `polygonMode` **must** be a valid `VkPolygonMode` value

- **VUID-VkPipelineRasterizationStateCreateInfo-cullMode-parameter**
  
  `cullMode` **must** be a valid combination of `VkCullModeFlagBits` values

- **VUID-VkPipelineRasterizationStateCreateInfo-frontFace-parameter**
  
  `frontFace` **must** be a valid `VkFrontFace` value

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineRasterizationStateCreateFlags;
```

`VkPipelineRasterizationStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.

If the `pNext` chain of `VkPipelineRasterizationStateCreateInfo` includes a `VkPipelineRasterizationDepthClipStateCreateInfoEXT` structure, then that structure controls whether depth clipping is enabled or disabled.

The `VkPipelineRasterizationDepthClipStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_depth_clip_enable
typedef struct VkPipelineRasterizationDepthClipStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationDepthClipStateCreateFlagsEXT flags;
    VkBool32 depthClipEnable;
} VkPipelineRasterizationDepthClipStateCreateInfoEXT;
```
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is reserved for future use.
• **depthClipEnable** controls whether depth clipping is enabled as described in *Primitive Clipping*.

### Valid Usage (Implicit)

- VUID-VkPipelineRasterizationDepthClipStateCreateInfoEXT-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT**
- VUID-VkPipelineRasterizationDepthClipStateCreateInfoEXT-flags-zerobitmask
  - **flags** must be 0

// Provided by VK_EXT_depth_clip_enable
typedef VkFlags VkPipelineRasterizationDepthClipStateCreateFlagsEXT;

**VkPipelineRasterizationDepthClipStateCreateFlagsEXT** is a bitmask type for setting a mask, but is currently reserved for future use.

The ** VkPipelineMultisampleStateCreateInfo** structure is defined as:

// Provided by VK_VERSION_1_0
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineMultisampleStateCreateFlags flags;
    VkSampleCountFlagBits rasterizationSamples;
    VkBool32 sampleShadingEnable;
    float minSampleShading;
    const VkSampleMask* pSampleMask;
    VkBool32 alphaToCoverageEnable;
    VkBool32 alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is reserved for future use.
• **rasterizationSamples** is a **VkSampleCountFlagBits** specifying the number of samples used in rasterization.
• **sampleShadingEnable** can be used to enable *Sample Shading*.
• **minSampleShading** specifies a minimum fraction of sample shading if **sampleShadingEnable** is set to **VK_TRUE**.
• `pSampleMask` is an array of `VkSampleMask` values used in the sample mask test.

• `alphaToCoverageEnable` controls whether a temporary coverage value is generated based on the alpha component of the fragment’s first color output as specified in the Multisample Coverage section.

• `alphaToOneEnable` controls whether the alpha component of the fragment’s first color output is replaced with one as described in Multisample Coverage.

Each bit in the sample mask is associated with a unique sample index as defined for the coverage mask. Each bit \( b \) for mask word \( w \) in the sample mask corresponds to sample index \( i \), where \( i = 32 \times w + b \). `pSampleMask` has a length equal to \( \lceil \frac{\text{rasterizationSamples}}{32} \rceil \) words.

If `pSampleMask` is `NULL`, it is treated as if the mask has all bits set to `1`.

Valid Usage

- **VUID-VkPipelineMultisampleStateCreateInfo-sampleShadingEnable-00784**
  If the sample rate shading feature is not enabled, `sampleShadingEnable` must be `VK_FALSE`

- **VUID-VkPipelineMultisampleStateCreateInfo-alphaToOneEnable-00785**
  If the alpha to one feature is not enabled, `alphaToOneEnable` must be `VK_FALSE`

- **VUID-VkPipelineMultisampleStateCreateInfo-minSampleShading-00786**
  `minSampleShading` must be in the range \([0,1]\)

- **VUID-VkPipelineMultisampleStateCreateInfo-rasterizationSamples-01415**
  If the `VK_NV_framebuffer_mixed_samples` extension is enabled, and if the subpass has any color attachments and `rasterizationSamples` is greater than the number of color samples, then `sampleShadingEnable` must be `VK_FALSE`
Valid Usage (Implicit)

- **VUID-VkPipelineMultisampleStateCreateInfo-sType-sType**
  - The `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO`.

- **VUID-VkPipelineMultisampleStateCreateInfo-pNext-pNext**
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkPipelineCoverageModulationStateCreateInfoNV`, `VkPipelineCoverageReductionStateCreateInfoNV`, `VkPipelineCoverageToColorStateCreateInfoNV`, or `VkPipelineSampleLocationsStateCreateInfoEXT`.

- **VUID-VkPipelineMultisampleStateCreateInfo-sType-unique**
  - The `sType` value of each struct in the `pNext` chain must be unique.

- **VUID-VkPipelineMultisampleStateCreateInfo-flags-zerobitmask**
  - The `flags` must be 0.

- **VUID-VkPipelineMultisampleStateCreateInfo-rasterizationSamples-parameter**
  - The `rasterizationSamples` must be a valid `VkSampleCountFlagBits` value.

- **VUID-VkPipelineMultisampleStateCreateInfo-pSampleMask-parameter**
  - If `pSampleMask` is not `NULL`, `pSampleMask` must be a valid pointer to an array of `VkSampleMask` values.

---

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineMultisampleStateCreateFlags;
```

`VkPipelineMultisampleStateCreateFlags` is a bitmask type for setting a mask, but it is currently reserved for future use.

The elements of the sample mask array are of type `VkSampleMask`, each representing 32 bits of coverage information.

```c
// Provided by VK_VERSION_1_0
typedef uint32_t VkSampleMask;
```

Rasterization only generates fragments which cover one or more pixels inside the framebuffer. Pixels outside the framebuffer are never considered covered in the fragment. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the pipeline, including any of the fragment operations.

Surviving fragments are processed by fragment shaders. Fragment shaders determine associated data for fragments, and can also modify or replace their assigned depth values.

When the `VK_AMD_mixed_attachment_samples` and `VK_NV_framebuffer_mixed_samples` extensions are not enabled, if the subpass for which this pipeline is being created uses color and/or depth/stencil rendering, the `VkSampleMask` values for the fragment may include additional information about the coverage of individual pixels within the fragment. This additional information is used by the fragment shader to determine which pixels within the fragment are considered to be covered, and to generate the appropriate depth or stencil values for those pixels.
attachments, then \textit{rasterizationSamples} must be the same as the sample count for those subpass attachments.

When the \texttt{VK_AMD\_mixed\_attachment\_samples} extension is enabled, if the subpass for which this pipeline is being created uses color and/or depth/stencil attachments, then \textit{rasterizationSamples} must be the same as the maximum of the sample counts of those subpass attachments.

When the \texttt{VK\_NV\_framebuffer\_mixed\_samples} extension is enabled, \textit{rasterizationSamples} must match the sample count of the depth/stencil attachment if present, otherwise must be greater than or equal to the sample count of the color attachments, if present.

If the \texttt{VK\_NV\_coverage\_reduction\_mode} extension is enabled, the coverage reduction mode specified by \texttt{VkPipelineCoverageReductionStateCreateInfoNV::coverageReductionMode}, the \textit{rasterizationSamples} member of \texttt{pMultisampleState} and the sample counts for the color and depth/stencil attachments (if present) must be a valid combination returned by \texttt{vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV}.

If the subpass for which this pipeline is being created does not use color or depth/stencil attachments, \textit{rasterizationSamples} must follow the rules for a zero-attachment subpass.

\section*{27.1. Discarding Primitives Before Rasterization}

Primitives are discarded before rasterization if the \textit{rasterizerDiscardEnable} member of \texttt{VkPipelineRasterizationStateCreateInfo} is enabled. When enabled, primitives are discarded after they are processed by the last active shader stage in the pipeline before rasterization.

To dynamically control whether primitives are discarded before the rasterization stage:

\begin{verbatim}
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetRasterizerDiscardEnableEXT(
    VkCommandBuffer commandBuffer, 
    VkBool32 rasterizerDiscardEnable);
\end{verbatim}

- \texttt{commandBuffer} is the command buffer into which the command will be recorded.
- \texttt{rasterizerDiscardEnable} controls whether primitives are discarded immediately before the rasterization stage.

This command sets the state for a given draw when the graphics pipeline is created with \texttt{VK\_DYNAMIC\_STATE\_RASTERIZER\_DISCARD\_ENABLE\_EXT} set in \texttt{VkPipelineDynamicStateCreateInfo::pDynamicStates}.

\begin{table}[h]
\centering
\begin{tabular}{|c|}
\hline
\textbf{Valid Usage} \\
\hline
- \texttt{VUID-vkCmdSetRasterizerDiscardEnableEXT-None-04871} \\
  \texttt{The extendedDynamicState2 feature} \textbf{must} be \textbf{enabled} \\
\hline
\end{tabular}
\end{table}
27.2. Controlling the Vertex Stream Used for Rasterization

By default vertex data output from the last vertex processing stage are directed to vertex stream zero. Geometry shaders can emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams. As vertices are received on each vertex stream, they are arranged into primitives of the type specified by the geometry shader output primitive type. The shading language instructions `OpEndPrimitive` and `OpEndStreamPrimitive` can be used to end the primitive being assembled on a given vertex stream and start a new empty primitive of the same type. An implementation supports up to `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams` streams, which is at least 1. The individual streams are numbered 0 through `maxTransformFeedbackStreams` minus 1. There is no requirement on the order of the streams to which vertices are emitted, and the number of vertices emitted to each vertex stream can be completely independent, subject only to the `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreamDataSize` and `VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferDataSize` limits. The primitives output from all vertex streams are passed to the transform feedback stage to be captured to transform feedback buffers in the manner specified by the last vertex processing stage shader’s
XfbBuffer, XfbStride, and Offsets decorations on the output interface variables in the graphics pipeline. To use a vertex stream other than zero, or to use multiple streams, the GeometryStreams capability must be specified.

By default, the primitives output from vertex stream zero are rasterized. If the implementation supports the \texttt{VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackRasterizationStreamSelect} property it is possible to rasterize a vertex stream other than zero.

By default, geometry shaders that emit vertices to multiple vertex streams are limited to using only the OutputPoints output primitive type. If the implementation supports the \texttt{VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackStreamsLinesTriangles} property it is possible to emit OutputLineStrip or OutputTriangleStrip in addition to OutputPoints.

The vertex stream used for rasterization is specified by adding a \texttt{VkPipelineRasterizationStateStreamCreateInfoEXT} structure to the pNext chain of a \texttt{VkPipelineRasterizationStateCreateInfo} structure.

The \texttt{VkPipelineRasterizationStateStreamCreateInfoEXT} structure is defined as:

```c
// Provided by VK_EXT_transform_feedback
typedef struct VkPipelineRasterizationStateStreamCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationStateStreamCreateFlagsEXT flags;
    uint32_t rasterizationStream;
} VkPipelineRasterizationStateStreamCreateInfoEXT;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is NULL or a pointer to a structure extending this structure.
- \texttt{flags} is reserved for future use.
- \texttt{rasterizationStream} is the vertex stream selected for rasterization.

If this structure is not present, \texttt{rasterizationStream} is assumed to be zero.

Valid Usage

- \texttt{VUID-VkPipelineRasterizationStateStreamCreateInfoEXT-geometryStreams-02324} \texttt{VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams} must be enabled
- \texttt{VUID-VkPipelineRasterizationStateStreamCreateInfoEXT-rasterizationStream-02325} \texttt{rasterizationStream} must be less than \texttt{VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams}
- \texttt{VUID-VkPipelineRasterizationStateStreamCreateInfoEXT-rasterizationStream-02326} \texttt{rasterizationStream} must be zero if \texttt{VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackRasterizationStreamSelect} is VK_FALSE
Valid Usage (Implicit)

- VUID-VkPipelineRasterizationStateStreamCreateInfoEXT-sType-sType
  sType must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_STREAM_CREATE_INFO_EXT`

- VUID-VkPipelineRasterizationStateStreamCreateInfoEXT-flags-zerobitmask
  flags must be 0

// Provided by VK_EXT_transform_feedback
typedef VkFlags VkPipelineRasterizationStateStreamCreateFlagsEXT;

VkPipelineRasterizationStateStreamCreateFlagsEXT is a bitmask type for setting a mask, but is currently reserved for future use.

27.3. Rasterization Order

Within a subpass of a render pass instance, for a given (x,y,layer,sample) sample location, the following operations are guaranteed to execute in rasterization order, for each separate primitive that includes that sample location:

1. Fragment operations, in the order defined
2. Blending, logic operations, and color writes

Each operation is atomically executed for each primitive and sample location.

Execution of these operations for each primitive in a subpass occurs in an order determined by the application.

The rasterization order to use for a graphics pipeline is specified by adding a `VkPipelineRasterizationStateRasterizationOrderAMD` structure to the `pNext` chain of a `VkPipelineRasterizationStateCreateInfo` structure.

The `VkPipelineRasterizationStateRasterizationOrderAMD` structure is defined as:

// Provided by VK_AMD_rasterization_order
typedef struct VkPipelineRasterizationStateRasterizationOrderAMD {
    VkStructureType            sType;
    const void*                pNext;
    VkRasterizationOrderAMD    rasterizationOrder;
} VkPipelineRasterizationStateRasterizationOrderAMD;

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `rasterizationOrder` is a `VkRasterizationOrderAMD` value specifying the primitive rasterization order to use.
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_RASTERIZATION_ORDER_AMD`
- `rasterizationOrder` must be a valid `VkRasterizationOrderAMD` value

If the `VK_AMD_rasterization_order` device extension is not enabled or the application does not request a particular rasterization order through specifying a `VkPipelineRasterizationStateRasterizationOrderAMD` structure then the rasterization order used by the graphics pipeline defaults to `VK_RASTERIZATION_ORDER_STRICT_AMD`.

Possible values of `VkPipelineRasterizationStateRasterizationOrderAMD::rasterizationOrder`, specifying the primitive rasterization order, are:

```c
typedef enum VkRasterizationOrderAMD {
    VK_RASTERIZATION_ORDER_STRICT_AMD = 0,
    VK_RASTERIZATION_ORDER_RELAXED_AMD = 1,
} VkRasterizationOrderAMD;
```

- `VK_RASTERIZATION_ORDER_STRICT_AMD` specifies that operations for each primitive in a subpass must occur in primitive order.
- `VK_RASTERIZATION_ORDER_RELAXED_AMD` specifies that operations for each primitive in a subpass may not occur in primitive order.

### 27.4. Multisampling

Multisampling is a mechanism to antialias all Vulkan primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. Each sample in each framebuffer attachment has storage for a color, depth, and/or stencil value, such that per-fragment operations apply to each sample independently. The color sample values can be later resolved to a single color (see Resolving Multisample Images and the Render Pass chapter for more details on how to resolve multisample images to non-multisample images).

Vulkan defines rasterization rules for single-sample modes in a way that is equivalent to a multisample mode with a single sample in the center of each fragment.

Each fragment includes a coverage mask with a single bit for each sample in the fragment, and a number of depth values and associated data for each sample. An implementation may choose to assign the same associated data to more than one sample. The location for evaluating such associated data may be anywhere within the fragment area including the fragment's center location \((x_f, y_f)\) or any of the sample locations. When `rasterizationSamples` is `VK_SAMPLE_COUNT_1_BIT`, the fragment's center location must be used. The different associated data values need not all be evaluated at the same location.
It is understood that each pixel has \texttt{rasterizationSamples} locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel \textbf{must} be located inside or on the boundary of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points \textbf{may} be identical for each pixel in the framebuffer, or they \textbf{may} differ.

If the render pass has a fragment density map attachment, each fragment only has \texttt{rasterizationSamples} locations associated with it regardless of how many pixels are covered in the fragment area. Fragment sample locations are defined as if the fragment had an area of (1,1) and its sample points \textbf{must} be located within these bounds. Their actual location in the framebuffer is calculated by scaling the sample location by the fragment area. Attachments with storage for multiple samples per pixel are located at the pixel sample locations. Otherwise, the fragment's sample locations are generally used for evaluation of associated data and fragment operations.

If the current pipeline includes a fragment shader with one or more variables in its interface decorated with \texttt{Sample} and \texttt{Input}, the data associated with those variables will be assigned independently for each sample. The values for each sample \textbf{must} be evaluated at the location of the sample. The data associated with any other variables not decorated with \texttt{Sample} and \texttt{Input} need not be evaluated independently for each sample.

A \textit{coverage mask} is generated for each fragment, based on which samples within that fragment are determined to be within the area of the primitive that generated the fragment.

Single pixel fragments and multi-pixel fragments defined by a \textit{fragment density map} have one set of samples. Multi-pixel fragments defined by a \textit{shading rate image} have one set of samples per pixel. Multi-pixel fragments defined by setting the \textit{fragment shading rate} have one set of samples per pixel. Each set of samples has a number of samples determined by \texttt{VkPipelineMultisampleStateCreateInfo}::\texttt{rasterizationSamples}. Each sample in a set is assigned a unique \textit{sample index} \(i\) in the range \([0, \texttt{rasterizationSamples})\). Each sample in a fragment is also assigned a unique \textit{coverage index} \(j\) in the range \([0, n \times \texttt{rasterizationSamples})\), where \(n\) is the number of sets in the fragment. If the fragment contains a single set of samples, the \textit{coverage index} is always equal to the \textit{sample index}. If a \textit{shading rate image} is used and a fragment covers multiple pixels, the coverage index is determined as defined by \texttt{VkPipelineViewportCoarseSampleOrderStateCreateInfoNV} or \texttt{vkCmdSetCoarseSampleOrderNV}.

If the \textit{fragment shading rate} is set, the coverage index \(j\) is determined as a function of the \textit{pixel index} \(p\), the \textit{sample index} \(i\), and the number of rasterization samples \(r\) as:

\[ j = i + r \times ((f_w \times f_h) - 1 - p) \]

where the pixel index \(p\) is determined as a function of the pixel's framebuffer location \((x,y)\) and the fragment size \((f_w,f_h)\):

\[ p_x = x \% f_w \]
\[ p_y = y \% f \]
\[ p = p_x + (p_y \times f_w) \]

The table below illustrates the pixel index for multi-pixel fragments:

**Table 34. Pixel indices - 1 wide**

<table>
<thead>
<tr>
<th>1x1</th>
<th>1x2</th>
<th>1x4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Table 35. Pixel indices - 2 wide**

<table>
<thead>
<tr>
<th>2x1</th>
<th>2x2</th>
<th>2x4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Table 36. Pixel indices - 4 wide**

<table>
<thead>
<tr>
<th>4x1</th>
<th>4x2</th>
<th>4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
<td><img src="image9" alt="Diagram" /></td>
</tr>
</tbody>
</table>
The coverage mask includes \( B \) bits packed into \( W \) words, defined as:

\[
B = n \times \text{rasterizationSamples}
\]

\[
W = \lceil B/32 \rceil
\]

Bit \( b \) in coverage mask word \( w \) is 1 if the sample with coverage index \( j = 32w + b \) is covered, and 0 otherwise.

If the \textit{standardSampleLocations} member of \textit{VkPhysicalDeviceLimits} is \textit{VK_TRUE}, then the sample counts \textit{VK_SAMPLE_COUNT_1_BIT}, \textit{VK_SAMPLE_COUNT_2_BIT}, \textit{VK_SAMPLE_COUNT_4_BIT}, \textit{VK_SAMPLE_COUNT_8_BIT}, and \textit{VK_SAMPLE_COUNT_16_BIT} have sample locations as listed in the following table, with the \( i \)th entry in the table corresponding to sample index \( i \). \textit{VK_SAMPLE_COUNT_32_BIT} and \textit{VK_SAMPLE_COUNT_64_BIT} do not have standard sample locations. Locations are defined relative to an origin in the upper left corner of the fragment.
Color images created with multiple samples per pixel use a compression technique where there are
two arrays of data associated with each pixel. The first array contains one element per sample where each element stores an index to the second array defining the fragment mask of the pixel. The second array contains one element per color fragment and each element stores a unique color value in the format of the image. With this compression technique it is not always necessary to actually use unique storage locations for each color sample: when multiple samples share the same color value the fragment mask may have two samples referring to the same color fragment. The number of color fragments is determined by the samples member of the VkImageCreateInfo structure used to create the image. The VK_AMD_shader_fragment_mask device extension provides shader instructions enabling the application to get direct access to the fragment mask and the individual color fragment values.

![Fragment Mask Diagram]

Figure 16. Fragment Mask

27.5. Custom Sample Locations

Applications can also control the sample locations used for rasterization.

If the pNext chain of the VkPipelineMultisampleStateCreateInfo structure specified at pipeline creation time includes a VkPipelineSampleLocationsStateCreateInfoEXT structure, then that structure controls the sample locations used when rasterizing primitives with the pipeline.

The VkPipelineSampleLocationsStateCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkPipelineSampleLocationsStateCreateInfoEXT {
    VkStructureType             sType;
    const void*                 pNext;
    VkBool32                    sampleLocationsEnable;
    VkSampleLocationsInfoEXT    sampleLocationsInfo;
} VkPipelineSampleLocationsStateCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **sampleLocationsEnable** controls whether custom sample locations are used. If sampleLocationsEnable is VK_FALSE, the default sample locations are used and the values specified in sampleLocationsInfo are ignored.
- **sampleLocationsInfo** is the sample locations to use during rasterization if sampleLocationsEnable is VK_TRUE and the graphics pipeline is not created with VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT.
Valid Usage (Implicit)

- VUID-VkPipelineSampleLocationsStateCreateInfoEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT`

- VUID-VkPipelineSampleLocationsStateCreateInfoEXT-sampleLocationsInfo-parameter
  
  `sampleLocationsInfo` must be a valid `VkSampleLocationsInfoEXT` structure

The `VkSampleLocationsInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkSampleLocationsInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkSampleCountFlagBits sampleLocationsPerPixel;
    VkExtent2D sampleLocationGridSize;
    uint32_t sampleLocationsCount;
    const VkSampleLocationEXT* pSampleLocations;
} VkSampleLocationsInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `sampleLocationsPerPixel` is a `VkSampleCountFlagBits` value specifying the number of sample locations per pixel.
- `sampleLocationGridSize` is the size of the sample location grid to select custom sample locations for.
- `sampleLocationsCount` is the number of sample locations in `pSampleLocations`.
- `pSampleLocations` is a pointer to an array of `sampleLocationsCount` `VkSampleLocationEXT` structures.

This structure can be used either to specify the sample locations to be used for rendering or to specify the set of sample locations an image subresource has been last rendered with for the purposes of layout transitions of depth/stencil images created with `VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT`.

The sample locations in `pSampleLocations` specify `sampleLocationsPerPixel` number of sample locations for each pixel in the grid of the size specified in `sampleLocationGridSize`. The sample location for sample `i` at the pixel grid location `(x,y)` is taken from `pSampleLocations[(x + y × sampleLocationGridSize.width) × sampleLocationsPerPixel + i]`.

If the render pass has a fragment density map, the implementation will choose the sample locations for the fragment and the contents of `pSampleLocations` may be ignored.
Valid Usage

- **VUID-VkSampleLocationsInfoEXT-sampleLocationsPerPixel-01526**
  
  `sampleLocationsPerPixel` must be a bit value that is set in `VkPhysicalDeviceSampleLocationsPropertiesEXT::sampleLocationSampleCounts`

- **VUID-VkSampleLocationsInfoEXT-sampleLocationsCount-01527**
  
  `sampleLocationsCount` must equal `sampleLocationsPerPixel × sampleLocationGridSize.width × sampleLocationGridSize.height`

Valid Usage (Implicit)

- **VUID-VkSampleLocationsInfoEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT`

- **VUID-VkSampleLocationsInfoEXT-pSampleLocations-parameter**
  
  If `sampleLocationsCount` is not 0, `pSampleLocations` must be a valid pointer to an array of `sampleLocationsCount` `VkSampleLocationEXT` structures

The `VkSampleLocationEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkSampleLocationEXT {
    float x;
    float y;
} VkSampleLocationEXT;
```

- `x` is the horizontal coordinate of the sample's location.
- `y` is the vertical coordinate of the sample's location.

The domain space of the sample location coordinates has an upper-left origin within the pixel in framebuffer space.

The values specified in a `VkSampleLocationEXT` structure are always clamped to the implementation-dependent sample location coordinate range `[sampleLocationCoordinateRange[0], sampleLocationCoordinateRange[1]]` that can be queried using `VkPhysicalDeviceSampleLocationsPropertiesEXT`.

The custom sample locations used for rasterization when `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` is `VK_TRUE` are specified by the `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsInfo` property of the bound graphics pipeline, if the pipeline was not created with `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT` enabled.

Otherwise, the sample locations used for rasterization are set by calling `vkCmdSetSampleLocationsEXT`:
// Provided by VK_EXT_sample_locations
void vkCmdSetSampleLocationsEXT(
    VkCommandBuffer                             commandBuffer,
    const VkSampleLocationsInfoEXT*             pSampleLocationsInfo);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pSampleLocationsInfo` is the sample locations state to set.

**Valid Usage**

- VUID-vkCmdSetSampleLocationsEXT-sampleLocationsPerPixel-01529
  The `sampleLocationsPerPixel` member of `pSampleLocationsInfo` must equal the `rasterizationSamples` member of the `VkPipelineMultisampleStateCreateInfo` structure the bound graphics pipeline has been created with.

- VUID-vkCmdSetSampleLocationsEXT-variableSampleLocations-01530
  If `VkPhysicalDeviceSampleLocationsPropertiesEXT::variableSampleLocations` is `VK_FALSE` then the current render pass must have been begun by specifying a `VkRenderPassSampleLocationsBeginInfoEXT` structure whose `pPostSubpassSampleLocations` member contains an element with a `subpassIndex` matching the current subpass index and the `sampleLocationsInfo` member of that element must match the sample locations state pointed to by `pSampleLocationsInfo`.

**Valid Usage (Implicit)**

- VUID-vkCmdSetSampleLocationsEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- VUID-vkCmdSetSampleLocationsEXT-pSampleLocationsInfo-parameter
  `pSampleLocationsInfo` must be a valid pointer to a valid `VkSampleLocationsInfoEXT` structure.

- VUID-vkCmdSetSampleLocationsEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state.

- VUID-vkCmdSetSampleLocationsEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized.
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.
27.6. Fragment Shading Rates

The features advertised by `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` allow an application to control the shading rate of a given fragment shader invocation.

The fragment shading rate strongly interacts with Multisampling, and the set of available rates for an implementation may be restricted by sample rate.

To query available shading rates, call:

```c
// Provided by VK_KHR_fragment_shading_rate
VkResult vkGetPhysicalDeviceFragmentShadingRatesKHR(
    VkPhysicalDevice physicalDevice,  // Provided by VK_KHR_fragment_shading_rate
    uint32_t* pFragmentShadingRateCount, // Provided by VK_KHR_fragment_shading_rate
    VkPhysicalDeviceFragmentShadingRateKHR* pFragmentShadingRates); // Provided by VK_KHR_fragment_shading_rate
```

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pFragmentShadingRateCount` is a pointer to an integer related to the number of fragment shading rates available or queried, as described below.
- `pFragmentShadingRates` is either `NULL` or a pointer to an array of `VkPhysicalDeviceFragmentShadingRateKHR` structures.

If `pFragmentShadingRates` is `NULL`, then the number of fragment shading rates available is returned in `pFragmentShadingRateCount`. Otherwise, `pFragmentShadingRateCount` must point to a variable set by the user to the number of elements in the `pFragmentShadingRates` array, and on return the variable is overwritten with the number of structures actually written to `pFragmentShadingRates`. If `pFragmentShadingRateCount` is less than the number of fragment shading rates available, at most `pFragmentShadingRateCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available fragment shading rates were returned.

The returned array of fragment shading rates must be ordered from largest `fragmentSize.width` value to smallest, and each set of fragment shading rates with the same `fragmentSize.width` value must be ordered from largest `fragmentSize.height` to smallest. Any two entries in the array must not have the same `fragmentSize` values.

For any entry in the array, the following rules also apply:

- The value of `fragmentSize.width` must be less than or equal to `maxFragmentSize.width`. 
• The value of `fragmentSize.width` must be greater than or equal to 1.
• The value of `fragmentSize.width` must be a power-of-two.
• The value of `fragmentSize.height` must be less than or equal to `maxFragmentSize.height`.
• The value of `fragmentSize.height` must be greater than or equal to 1.
• The value of `fragmentSize.height` must be a power-of-two.
• The highest sample count in `sampleCounts` must be less than or equal to `maxFragmentShadingRateRasterizationSamples`.
• The product of `fragmentSize.width`, `fragmentSize.height`, and the highest sample count in `sampleCounts` must be less than or equal to `maxFragmentShadingRateCoverageSamples`.

Implementations must support at least the following shading rates:

<table>
<thead>
<tr>
<th><code>sampleCounts</code></th>
<th><code>fragmentSize</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>`VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT`</td>
</tr>
<tr>
<td>`VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT`</td>
</tr>
<tr>
<td>~0</td>
<td>{1,1}</td>
</tr>
</tbody>
</table>

If `framebufferColorSampleCounts` includes `VK_SAMPLE_COUNT_2_BIT`, the required rates must also include `VK_SAMPLE_COUNT_2_BIT`.

Note

Including the {1,1} fragment size is done for completeness; it has no actual effect on the support of rendering without setting the fragment size. All sample counts and render pass transforms are supported for this rate.

The returned set of fragment shading rates must be returned in the native (rotated) coordinate system. For rasterization using render pass transform not equal to `VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR`, the application must transform the returned fragment shading rates into the current (unrotated) coordinate system to get the supported rates for that transform.

Note

For example, consider an implementation returning support for 4x2, but not 2x4 in the set of supported fragment shading rates. This means that for transforms `VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR` and `VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR`, 2x4 is a supported rate, but 4x2 is an unsupported rate.
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceFragmentShadingRatesKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceFragmentShadingRatesKHR-pFragmentShadingRateCount-parameter
  pFragmentShadingRateCount must be a valid pointer to a uint32_t value

- VUID-vkGetPhysicalDeviceFragmentShadingRatesKHR-pFragmentShadingRates-parameter
  If the value referenced by pFragmentShadingRateCount is not 0, and
  pFragmentShadingRates is not NULL, pFragmentShadingRates must be a valid pointer to an array of
  pFragmentShadingRateCount VkPhysicalDeviceFragmentShadingRateKHR structures

Return Codes

Success
- VK SUCCESS
- VK_INCOMPLETE

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY

The VkPhysicalDeviceFragmentShadingRateKHR structure is defined as

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPhysicalDeviceFragmentShadingRateKHR {
    VkStructureType       sType;
    void*                 pNext;
    VkSampleCountFlags    sampleCounts;
    VkExtent2D            fragmentSize;
} VkPhysicalDeviceFragmentShadingRateKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- sampleCounts is a bitmask of sample counts for which the shading rate described by fragmentSize is supported.
- fragmentSize is a VkExtent2D describing the width and height of a supported shading rate.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceFragmentShadingRateKHR-sType-sType**
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_KHR
- **VUID-VkPhysicalDeviceFragmentShadingRateKHR-pNext-pNext**
  - pNext must be NULL

Fragment shading rates can be set at three points, with the three rates combined to determine the final shading rate.

### 27.6.1. Pipeline Fragment Shading Rate

The pipeline fragment shading rate can be set on a per-draw basis by either setting the rate in a graphics pipeline, or dynamically via `vkCmdSetFragmentShadingRateKHR`.

The `VkPipelineFragmentShadingRateStateCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPipelineFragmentShadingRateStateCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkExtent2D fragmentSize;
    VkFragmentShadingRateCombinerOpKHR combinerOps[2];
} VkPipelineFragmentShadingRateStateCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **fragmentSize** specifies a `VkExtent2D` structure containing the fragment size used to define the pipeline fragment shading rate for draw commands using this pipeline.
- **combinerOps** specifies a `VkFragmentShadingRateCombinerOpKHR` value determining how the pipeline, primitive, and attachment shading rates are combined for fragments generated by drawing commands using the created pipeline.

If the **pNext** chain of `VkGraphicsPipelineCreateInfo` includes a `VkPipelineFragmentShadingRateStateCreateInfoKHR` structure, then that structure includes parameters that control the pipeline fragment shading rate.

If this structure is not present, **fragmentSize** is considered to be equal to (1,1), and both elements of **combinerOps** are considered to be equal to **VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR**.
Valid Usage (Implicit)

- **VUID-VkPipelineFragmentShadingRateStateCreateInfoKHR-sType-sType**
  
  **sType** must be `VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_STATE_CREATE_INFO_KHR`

- **VUID-VkPipelineFragmentShadingRateStateCreateInfoKHR-combinerOps-parameter**
  
  Any given element of `combinerOps` must be a valid `VkFragmentShadingRateCombinerOpKHR` value.

If a pipeline state object is created with `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` enabled, the pipeline fragment shading rate and combiner operation is set by the command:

```c
// Provided by VK_KHR_fragment_shading_rate
void vkCmdSetFragmentShadingRateKHR(
    VkCommandBuffer commandBuffer,
    const VkExtent2D* pFragmentSize,
    const VkFragmentShadingRateCombinerOpKHR combinerOps[2]);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **pFragmentSize** specifies the pipeline fragment shading rate for subsequent draw commands.
- **combinerOps** specifies a `VkFragmentShadingRateCombinerOpKHR` determining how the pipeline, primitive, and attachment shading rates are combined for fragments generated by subsequent drawing commands.
Valid Usage

- VUID-vkCmdSetFragmentShadingRateKHR-pipelineFragmentShadingRate-04507
  If `pipelineFragmentShadingRate` is not enabled, `pFragmentSize->width` must be 1

- VUID-vkCmdSetFragmentShadingRateKHR-pipelineFragmentShadingRate-04508
  If `pipelineFragmentShadingRate` is not enabled, `pFragmentSize->height` must be 1

- VUID-vkCmdSetFragmentShadingRateKHR-pipelineFragmentShadingRate-04509
  One of `pipelineFragmentShadingRate`, `primitiveFragmentShadingRate`, or `attachmentFragmentShadingRate` must be enabled

- VUID-vkCmdSetFragmentShadingRateKHR-primitiveFragmentShadingRate-04510
  If the `primitiveFragmentShadingRate` feature is not enabled, `combinerOps[0]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`

- VUID-vkCmdSetFragmentShadingRateKHR-attachmentFragmentShadingRate-04511
  If the `attachmentFragmentShadingRate` feature is not enabled, `combinerOps[1]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`

- VUID-vkCmdSetFragmentShadingRateKHR-fragmentSizeNonTrivialCombinerOps-04512
  If the `fragmentSizeNonTrivialCombinerOps` limit is not supported, elements of `combinerOps` must be either `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR` or `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR`

- VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04513
  `pFragmentSize->width` must be greater than or equal to 1

- VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04514
  `pFragmentSize->height` must be greater than or equal to 1

- VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04515
  `pFragmentSize->width` must be a power-of-two value

- VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04516
  `pFragmentSize->height` must be a power-of-two value

- VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04517
  `pFragmentSize->width` must be less than or equal to 4

- VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-04518
  `pFragmentSize->height` must be less than or equal to 4
Valid Usage (Implicit)

- **VUID-vkCmdSetFragmentShadingRateKHR-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdSetFragmentShadingRateKHR-pFragmentSize-parameter**
  
  `pFragmentSize` must be a valid pointer to a valid `VkExtent2D` structure.

- **VUID-vkCmdSetFragmentShadingRateKHR-combinerOps-parameter**
  
  Any given element of `combinerOps` must be a valid `VkFragmentShadingRateCombinerOpKHR` value.

- **VUID-vkCmdSetFragmentShadingRateKHR-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdSetFragmentShadingRateKHR-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

27.6.2. Primitive Fragment Shading Rate

The *primitive fragment shading rate* can be set via the `PrimitiveShadingRateKHR` built-in in the last active vertex processing shader stage. The rate associated with a given primitive is sourced from the value written to `PrimitiveShadingRateKHR` by that primitive’s *provoking vertex*.

27.6.3. Attachment Fragment Shading Rate

The *attachment shading rate* can be set by including `VkFragmentShadingRateAttachmentInfoKHR` in a subpass to define a *fragment shading rate attachment*. Each pixel in the framebuffer is assigned an attachment fragment shading rate by the corresponding texel in the fragment shading rate attachment, according to:
\[ x' = \text{floor}(x / \text{region}_x) \]

\[ y' = \text{floor}(y / \text{region}_y) \]

where \( x' \) and \( y' \) are the coordinates of a texel in the fragment shading rate attachment, \( x \) and \( y \) are the coordinates of the pixel in the framebuffer, and \( \text{region}_x \) and \( \text{region}_y \) are the size of the region each texel corresponds to, as defined by the `shadingRateAttachmentTexelSize` member of `VkFragmentShadingRateAttachmentInfoKHR`.

If `multiview` is enabled and the shading rate attachment has multiple layers, the shading rate attachment texel is selected from the layer determined by the `ViewIndex` built-in. If `multiview` is disabled, and both the shading rate attachment and the framebuffer have multiple layers, the shading rate attachment texel is selected from the layer determined by the `Layer` built-in. Otherwise, the texel is unconditionally selected from the first layer of the attachment.

The fragment size is encoded into the first component of the identified texel as follows:

\[ \text{size}_w = 2^{((\text{texel}/4) \& 3)} \]

\[ \text{size}_h = 2^{(\text{texel} \& 3)} \]

where texel is the value in the first component of the identified texel, and \( \text{size}_w \) and \( \text{size}_h \) are the width and height of the fragment size, decoded from the texel.

If no fragment shading rate attachment is specified, this size is calculated as \( \text{size}_w = \text{size}_h = 1 \).

Applications **must** not specify a width or height greater than 4 by this method.

The *Fragment Shading Rate* enumeration in SPIR-V adheres to the above encoding.

### 27.6.4. Combining the Fragment Shading Rates

The final rate \((C_{xy}')\) used for fragment shading **must** be one of the rates returned by `vkGetPhysicalDeviceFragmentShadingRatesKHR` for the sample count and render pass transform used by rasterization.

If any of the following conditions are met, \( C_{xy}' \) **must** be set to \( \{1,1\} \):

- If `Sample Shading` is enabled.
- The `fragmentShadingRateWithSampleMask` limit is not supported, and `VkPipelineMultisampleStateCreateInfo::pSampleMask` contains a zero value in any bit used by fragment operations.
- The `fragmentShadingRateWithShaderSampleMask` is not supported, and the fragment shader has `SampleMask` in the input or output interface.
- The `fragmentShadingRateWithShaderDepthStencilWrites` limit is not supported, and the fragment
shader declares the `FragDepth` or `FragStencilRefEXT` built-in.

- The `fragmentShadingRateWithConservativeRasterization` limit is not supported, and `VkPipelineRasterizationConservativeStateCreateInfoEXT::conservativeRasterizationMode` is not `VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT`.

- The `fragmentShadingRateWithFragmentShaderInterlock` limit is not supported, and the fragment shader declares any of the fragment shader interlock execution modes.

- The `fragmentShadingRateWithCustomSampleLocations` limit is not supported, and `VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable` is `VK_TRUE`.

Otherwise, each of the specified shading rates are combined and then used to derive the value of \(C_{xy}'\). As there are three ways to specify shading rates, two combiner operations are specified - between the pipeline and primitive shading rates, and between the result of that and the attachment shading rate.

The equation used for each combiner operation is defined by `VkFragmentShadingRateCombinerOpKHR`:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef enum VkFragmentShadingRateCombinerOpKHR {
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR = 0,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR = 1,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MIN_KHR = 2,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MAX_KHR = 3,
    VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR = 4,
} VkFragmentShadingRateCombinerOpKHR;
```

- `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR` specifies a combiner operation of \(\text{combine}(A_{xy}, B_{xy}) = A_{xy}\).

- `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR` specifies a combiner operation of \(\text{combine}(A_{xy}, B_{xy}) = B_{xy}\).

- `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MIN_KHR` specifies a combiner operation of \(\text{combine}(A_{xy}, B_{xy}) = \min(A_{xy}, B_{xy})\).

- `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MAX_KHR` specifies a combiner operation of \(\text{combine}(A_{xy}, B_{xy}) = \max(A_{xy}, B_{xy})\).

- `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR` specifies a combiner operation of \(\text{combine}(A_{xy}, B_{xy}) = A_{xy} \times B_{xy}\).

where `\text{combine}(A_{xy}, B_{xy})` is the combine operation, and \(A_{xy}\) and \(B_{xy}\) are the inputs to the operation.

If `fragmentShadingRateStrictMultiplyCombiner` is `VK_FALSE`, using `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR` with values of 1 for both \(A\) and \(B\) in the same dimension results in the value 2 being produced for that dimension. See the definition of `fragmentShadingRateStrictMultiplyCombiner` for more information.

These operations are performed in a component-wise fashion.

This is used to generate a combined fragment area using the equation:
\[ C_{xy} = \text{combine}(A_{xy}, B_{xy}) \]

where \( C_{xy} \) is the combined fragment area result, and \( A_{xy} \) and \( B_{xy} \) are the fragment areas of the fragment shading rates being combined.

Two combine operations are performed, first with \( A_{xy} \) equal to the pipeline fragment shading rate and \( B_{xy} \) equal to the primitive fragment shading rate, with the combine() operation selected by combinerOps[0]. A second combination is then performed, with \( A_{xy} \) equal to the result of the first combination and \( B_{xy} \) equal to the attachment fragment shading rate, with the combine() operation selected by combinerOps[1]. The result of the second combination is used as the final fragment shading rate, reported via the ShadingRateKHR built-in.

Implementations may clamp the \( C_{xy} \) result of each combiner operation separately, or only after the second combiner operation.

If the final combined rate is one of the rates returned by vkGetPhysicalDeviceFragmentShadingRatesKHR for the sample count and render pass transform used by rasterization, \( C_{xy}' = C_{xy} \). Otherwise, \( C_{xy}' \) is selected from the rates returned by vkGetPhysicalDeviceFragmentShadingRatesKHR for the sample count and render pass transform used by rasterization. From this list of supported rates, the following steps are applied in order, to select a single value:

1. Keep only rates where \( C_{x}' \leq C_{x} \) and \( C_{y}' \leq C_{y} \).
   - Implementations may also keep rates where \( C_{x}' \leq C_{y} \) and \( C_{y}' \leq C_{x} \).
2. Keep only rates with the highest area \( (C_{x}' \times C_{y}') \).
3. Keep only rates with the lowest aspect ratio \( (C_{x}' + C_{y}') \).
4. In cases where a wide (e.g. 4x1) and tall (e.g. 1x4) rate remain, the implementation may choose either rate. However, it must choose this rate consistently for the same shading rates, render pass transform, and combiner operations for the lifetime of the VkDevice.

### 27.6.5. Extended Fragment Shading Rates

The features advertised by VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV provide support for additional fragment shading rates beyond those specifying one fragment shader invocation covering all pixels in a fragment whose size is indicated by the fragment shading rate.

If the fragmentShadingRate Enums feature is enabled, fragment shading rates may be specified using the VkFragmentShadingRateNV enumerated type defined as:
// Provided by VK_NV_fragment_shading_rate Enums

typedef enum VkFragmentShadingRateNV {
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_PIXEL_NV = 0,
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_1X2_PIXELS_NV = 1,
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_2X1_PIXELS_NV = 4,
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_2X2_PIXELS_NV = 5,
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_2X4_PIXELS_NV = 6,
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_4X2 PIXELS_NV = 9,
    VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_4X4_PIXELS_NV = 10,
    VK_FRAGMENT_SHADING_RATE_2_INVOCATIONS_PER_PIXEL_NV = 11,
    VK_FRAGMENT_SHADING_RATE_4_INVOCATIONS_PER_PIXEL_NV = 12,
    VK_FRAGMENT_SHADING_RATE_8_INVOCATIONS_PER_PIXEL_NV = 13,
    VK_FRAGMENT_SHADING_RATE_16_INVOCATIONS_PER_PIXEL_NV = 14,
    VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV = 15,
} VkFragmentShadingRateNV;

- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_PIXEL_NV specifies a fragment size of 1x1 pixels.
- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_1X2_PIXELS_NV specifies a fragment size of 1x2 pixels.
- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_2X1_PIXELS_NV specifies a fragment size of 2x1 pixels.
- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_2X2_PIXELS_NV specifies a fragment size of 2x2 pixels.
- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_2X4_PIXELS_NV specifies a fragment size of 2x4 pixels.
- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_4X2_PIXELS_NV specifies a fragment size of 4x2 pixels.
- VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_4X4_PIXELS_NV specifies a fragment size of 4x4 pixels.
- VK_FRAGMENT_SHADING_RATE_2_INVOCATIONS_PER_PIXEL_NV specifies a fragment size of 1x1 pixels, with two fragment shader invocations per fragment.
- VK_FRAGMENT_SHADING_RATE_4_INVOCATIONS_PER_PIXEL_NV specifies a fragment size of 1x1 pixels, with four fragment shader invocations per fragment.
- VK_FRAGMENT_SHADING_RATE_8_INVOCATIONS_PER_PIXEL_NV specifies a fragment size of 1x1 pixels, with eight fragment shader invocations per fragment.
- VK_FRAGMENT_SHADING_RATE_16_INVOCATIONS_PER_PIXEL_NV specifies a fragment size of 1x1 pixels, with sixteen fragment shader invocations per fragment.
- VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV specifies that any portions of a primitive that use that shading rate should be discarded without invoking any fragment shader.

To use the shading rates VK_FRAGMENT_SHADING_RATE_2_INVOCATIONS_PER_PIXEL_NV, VK_FRAGMENT_SHADING_RATE_4_INVOCATIONS_PER_PIXEL_NV, and VK_FRAGMENT_SHADING_RATE_8_INVOCATIONS_PER_PIXEL_NV,
VK_FRAGMENT_SHADING_RATE_16_INVOCATIONS_PER_PIXEL_NV as a pipeline, primitive, or attachment shading rate, the supersampleFragmentShadingRates feature must be enabled. To use the shading rate VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV as a pipeline, primitive, or attachment shading rate, the noInvocationFragmentShadingRates feature must be enabled.

When using fragment shading rate enums, the pipeline fragment shading rate can be set on a per-draw basis by either setting the rate in a graphics pipeline, or dynamically via vkCmdSetFragmentShadingRateEnumNV.

The VkPipelineFragmentShadingRateEnumStateCreateInfoNV structure is defined as:

```c
// Provided by VK_NV_fragment_shading_rate_enums
typedef struct VkPipelineFragmentShadingRateEnumStateCreateInfoNV {
  VkStructureType sType;
  const void* pNext;
  VkFragmentShadingRateTypeNV shadingRateType;
  VkFragmentShadingRateNV shadingRate;
  VkFragmentShadingRateCombinerOpKHR combinerOps[2];
} VkPipelineFragmentShadingRateEnumStateCreateInfoNV;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- shadingRateType specifies a VkFragmentShadingRateTypeNV value indicating whether fragment shading rates are specified using fragment sizes or VkFragmentShadingRateNV enums.
- shadingRate specifies a VkFragmentShadingRateNV value indicating the pipeline fragment shading rate.
- combinerOps specifies VkFragmentShadingRateCombinerOpKHR values determining how the pipeline, primitive, and attachment shading rates are combined for fragments generated by drawing commands using the created pipeline.

If the pNext chain of VkGraphicsPipelineCreateInfo includes a VkPipelineFragmentShadingRateEnumStateCreateInfoNV structure, then that structure includes parameters that control the pipeline fragment shading rate.

If this structure is not present, shadingRateType is considered to be equal to VK_FRAGMENT_SHADING_RATE_TYPE_FRAGMENT_SIZE_NV, shadingRate is considered to be equal to VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_PIXEL_NV, and both elements of combinerOps are considered to be equal to VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR.
Valid Usage (Implicit)

- VUID-VkPipelineFragmentShadingRateEnumStateCreateInfoNV-sType-sType
  
  sType must be `VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_ENUM_STATE_CREATE_INFO_NV`

- VUID-VkPipelineFragmentShadingRateEnumStateCreateInfoNV-shadingRateType-parameter
  
  shadingRateType must be a valid `VkFragmentShadingRateTypeNV` value

- VUID-VkPipelineFragmentShadingRateEnumStateCreateInfoNV-shadingRate-parameter
  
  shadingRate must be a valid `VkFragmentShadingRateNV` value

- VUID-VkPipelineFragmentShadingRateEnumStateCreateInfoNV-combinerOps-parameter
  
  Any given element of combinerOps must be a valid `VkFragmentShadingRateCombinerOpKHR` value

The `VkFragmentShadingRateTypeNV` enumerated type specifies whether a graphics pipeline gets its pipeline fragment shading rates and combiners from the `VkPipelineFragmentShadingRateEnumStateCreateInfoNV` structure or the `VkPipelineFragmentShadingRateStateCreateInfoKHR` structure.

```c
// Provided by VK_NV_fragment_shading_rate_enums
typedef enum VkFragmentShadingRateTypeNV {
    VK_FRAGMENT_SHADING_RATE_TYPE_FRAGMENT_SIZE_NV = 0,
    VK_FRAGMENT_SHADING_RATE_TYPE_ENUMS_NV = 1,
} VkFragmentShadingRateTypeNV;
```

- `VK_FRAGMENT_SHADING_RATE_TYPE_FRAGMENT_SIZE_NV` specifies that a graphics pipeline should obtain its pipeline fragment shading rate and shading rate combiner state from the `VkPipelineFragmentShadingRateStateCreateInfoKHR` structure and that any state specified by the `VkPipelineFragmentShadingRateEnumStateCreateInfoNV` structure should be ignored.

- `VK_FRAGMENT_SHADING_RATE_TYPE_ENUMS_NV` specifies that a graphics pipeline should obtain its pipeline fragment shading rate and shading rate combiner state from the `VkPipelineFragmentShadingRateStateCreateInfoKHR` structure and that any state specified by the `VkPipelineFragmentShadingRateEnumStateCreateInfoNV` structure should be ignored.

If a pipeline state object is created with `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR` enabled, the pipeline fragment shading rate and combiner operation may be set by the command:

```c
// Provided by VK_NV_fragment_shading_rate_enums
void vkCmdSetFragmentShadingRateEnumNV(
    VkCommandBuffer                       commandBuffer,
    VkFragmentShadingRateNV              shadingRate,
    const VkFragmentShadingRateCombinerOpKHR combinerOps[2]);
```

- `commandBuffer` is the command buffer into which the command will be recorded.

- `shadingRate` specifies a `VkFragmentShadingRateNV` enum indicating the pipeline fragment...
shading rate for subsequent draw commands.

- `combinerOps` specifies a `VkFragmentShadingRateCombinerOpKHR` determining how the pipeline, primitive, and attachment shading rates are combined for fragments generated by subsequent drawing commands.

---

**Valid Usage**

- VUID-vkCmdSetFragmentShadingRateEnumNV-pipelineFragmentShadingRate-04576
  
  If `pipelineFragmentShadingRate` is not enabled, `shadingRate` must be `VK_FRAGMENT_SHADING_RATE_1_INVOCATION_PER_PIXEL_NV`

- VUID-vkCmdSetFragmentShadingRateEnumNV-supersampleFragmentShadingRates-04577
  
  If `supersampleFragmentShadingRates` is not enabled, `shadingRate` must not be `VK_FRAGMENT_SHADING_RATE_2_INVOCATIONS_PER_PIXEL_NV`, `VK_FRAGMENT_SHADING_RATE_4_INVOCATIONS_PER_PIXEL_NV`, `VK_FRAGMENT_SHADING_RATE_8_INVOCATIONS_PER_PIXEL_NV`, or `VK_FRAGMENT_SHADING_RATE_16_INVOCATIONS_PER_PIXEL_NV`

- VUID-vkCmdSetFragmentShadingRateEnumNV-noInvocationFragmentShadingRates-04578
  
  If `noInvocationFragmentShadingRates` is not enabled, `shadingRate` must not be `VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV`

- VUID-vkCmdSetFragmentShadingRateEnumNV-fragmentShadingRateEnums-04579
  
  `fragmentShadingRateEnums` must be enabled

- VUID-vkCmdSetFragmentShadingRateEnumNV-pipelineFragmentShadingRate-04580
  
  One of `pipelineFragmentShadingRate`, `primitiveFragmentShadingRate`, or `attachmentFragmentShadingRate` must be enabled

- VUID-vkCmdSetFragmentShadingRateEnumNV-primitiveFragmentShadingRate-04581
  
  If the `primitiveFragmentShadingRate` feature is not enabled, `combinerOps[0]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`

- VUID-vkCmdSetFragmentShadingRateEnumNV-attachmentFragmentShadingRate-04582
  
  If the `attachmentFragmentShadingRate` feature is not enabled, `combinerOps[1]` must be `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR`

- VUID-vkCmdSetFragmentShadingRateEnumNV-fragmentSizeNonTrivialCombinerOps-04583
  
  If the `fragmentSizeNonTrivialCombinerOps` limit is not supported, elements of `combinerOps` must be either `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR` or `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR`
Valid Usage (Implicit)

- **VUID-vkCmdSetFragmentShadingRateEnumNV-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetFragmentShadingRateEnumNV-shadingRate-parameter**
  
  `shadingRate` must be a valid `VkFragmentShadingRateNV` value

- **VUID-vkCmdSetFragmentShadingRateEnumNV-combinerOps-parameter**
  
  Any given element of `combinerOps` must be a valid `VkFragmentShadingRateCombinerOpKHR` value

- **VUID-vkCmdSetFragmentShadingRateEnumNV-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdSetFragmentShadingRateEnumNV-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the `supersampleFragmentShadingRates` or `noInvocationFragmentShadingRates` features are enabled, the behavior of the shading rate combiner operations is extended to support the shading rates enabled by those features. Primitive and attachment shading rate values are interpreted as `VkFragmentShadingRateNV` values and the behavior of the combiners is modified as follows:

- For `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MIN_KHR`, `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MAX_KHR`, and `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR`, if either `A_{xy}` or `B_{xy}` is `VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV`, `combine(A_{xy}, B_{xy})` produces a shading rate of `VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV`, regardless of the other input shading rate.

- For `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MIN_KHR`, `combine(A_{xy}, B_{xy})` produces a shading rate whose fragment size is the smaller of the fragment sizes of `A_{xy}` and `B_{xy}` and whose invocation count is the larger of the invocation counts of `A_{xy}` and `B_{xy}`.
• For `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MAX_KHR`, combine($A_{xy}, B_{xy}$) produces a shading rate whose fragment size is the larger of the fragment sizes of $A_{xy}$ and $B_{xy}$ and whose invocation count is the smaller of the invocation counts of $A_{xy}$ and $B_{xy}$.

• For `VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR`, combine($A_{xy}, B_{xy}$) produces a shading rate whose fragment size and invocation count is the product of the fragment sizes and invocation counts, respectively, of $A_{xy}$ and $B_{xy}$. If the resulting shading rate has both multiple pixels and multiple invocations per fragment, an implementation may adjust the shading rate by reducing both the pixel and invocation counts.

If the final shading rate from the combiners is `VK_FRAGMENT_SHADING_RATE_NO_INVOCATIONS_NV`, no fragments will be generated for any portion of a primitive using that shading rate.

If the final shading rate from the combiners specifies multiple fragment shader invocations per fragment, the fragment will be processed with multiple unique samples as in sample shading, where the total number the total number of invocations is taken from the shading rate and then clamped to the value of `totalSamples` used by sample shading and to the value of `maxFragmentShadingRateInvocationCount`.

27.7. Shading Rate Image

The shading rate image feature allows pipelines to use a shading rate image to control the fragment area and the minimum number of fragment shader invocations launched for each fragment. When the shading rate image is enabled, the rasterizer determines a base shading rate for each region of the framebuffer covered by a primitive by fetching a value from the shading rate image and translating it to a shading rate using a per-viewport shading rate palette. This base shading rate is then adjusted to derive a final shading rate. The final shading rate specifies the fragment area and fragment shader invocation count to use for fragments generated in the region.

If the `pNext` chain of `VkPipelineViewportStateCreateInfo` includes a `VkPipelineViewportShadingRateImageStateCreateInfoNV` structure, then that structure includes parameters that control the shading rate.

The `VkPipelineViewportShadingRateImageStateCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_shading_rate_image
typedef struct VkPipelineViewportShadingRateImageStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkBool32 shadingRateImageEnable;
    uint32_t viewportCount;
    const VkShadingRatePaletteNV* pShadingRatePalettes;
} VkPipelineViewportShadingRateImageStateCreateInfoNV;
```

• `sType` is the type of this structure.

• `pNext` is `NULL` or a pointer to a structure extending this structure.

• `shadingRateImageEnable` specifies whether shading rate image and palettes are used during rasterization.
• **viewportCount** specifies the number of per-viewport palettes used to translate values stored in shading rate images.

• **pShadingRatePalettes** is a pointer to an array of **VkShadingRatePaletteNV** structures defining the palette for each viewport. If the shading rate palette state is dynamic, this member is ignored.

If this structure is not present, **shadingRateImageEnable** is considered to be **VK_FALSE**, and the shading rate image and palettes are not used.

### Valid Usage

- VUID-VkPipelineViewportShadingRateImageStateCreateInfoNV-viewportCount-02054
  
  If the **multiple viewports** feature is not enabled, **viewportCount** must be 0 or 1

- VUID-VkPipelineViewportShadingRateImageStateCreateInfoNV-viewportCount-02055
  
  **viewportCount** must be less than or equal to **VkPhysicalDeviceLimits**:maxViewports

- VUID-VkPipelineViewportShadingRateImageStateCreateInfoNV-shadingRateImageEnable-02056
  
  If **shadingRateImageEnable** is **VK_TRUE**, **viewportCount** must be greater or equal to the **viewportCount** member of **VkPipelineViewportStateCreateInfo**

### Valid Usage (Implicit)

- VUID-VkPipelineViewportShadingRateImageStateCreateInfoNV-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SHADING_RATE_IMAGE_STATE_CREATE_INFO_NV**

When shading rate image usage is enabled in the bound pipeline, the pipeline uses a shading rate image specified by the command:

```c
// Provided by VK_NV_shading_rate_image
define void vkCmdBindShadingRateImageNV(
    VkCommandBuffer                             commandBuffer,
    VkImageView                                 imageView,
    VkImageLayout                               imageLayout);
```

- **commandBuffer** is the command buffer into which the command will be recorded.

- **imageView** is an image view handle specifying the shading rate image. **imageView** may be set to **VK_NULL_HANDLE**, which is equivalent to specifying a view of an image filled with zero values.

- **imageLayout** is the layout that the image subresources accessible from **imageView** will be in when the shading rate image is accessed.
Valid Usage

• VUID-vkCmdBindShadingRateImageNV-None-02058
  The shading rate image feature must be enabled

• VUID-vkCmdBindShadingRateImageNV-imageView-02059
  If imageView is not VK_NULL_HANDLE, it must be a valid VkImageView handle of type
  VK_IMAGE_VIEW_TYPE_2D or VK_IMAGE_VIEW_TYPE_2D_ARRAY

• VUID-vkCmdBindShadingRateImageNV-imageView-02060
  If imageView is not VK_NULL_HANDLE, it must have a format of VK_FORMAT_R8_UINT

• VUID-vkCmdBindShadingRateImageNV-imageView-02061
  If imageView is not VK_NULL_HANDLE, it must have been created with a usage value
  including VK_IMAGE_USAGE_SHADING_RATE_IMAGE_BIT_NV

• VUID-vkCmdBindShadingRateImageNV-imageView-02062
  If imageView is not VK_NULL_HANDLE, imageLayout must match the actual VkImageLayout
  of each subresource accessible from imageView at the time the subresource is accessed

• VUID-vkCmdBindShadingRateImageNV-imageLayout-02063
  If imageView is not VK_NULL_HANDLE, imageLayout must be
  VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV or VK_IMAGE_LAYOUT_GENERAL

Valid Usage (Implicit)

• VUID-vkCmdBindShadingRateImageNV-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdBindShadingRateImageNV-imageView-parameter
  If imageView is not VK_NULL_HANDLE, imageView must be a valid VkImageView handle

• VUID-vkCmdBindShadingRateImageNV-imageLayout-parameter
  imageLayout must be a valid VkImageLayout value

• VUID-vkCmdBindShadingRateImageNV-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdBindShadingRateImageNV-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics
  operations

• VUID-vkCmdBindShadingRateImageNV-commonparent
  Both of commandBuffer, and imageView that are valid handles of non-ignored parameters
  must have been created, allocated, or retrieved from the same VkDevice
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the shading rate image is enabled in the current pipeline, rasterizing a primitive covering the pixel with coordinates \((x,y)\) will fetch a shading rate index value from the shading rate image bound by `vkCmdBindShadingRateImageNV`. If the shading rate image view has a type of `VK_IMAGE_VIEW_TYPE_2D`, the lookup will use texel coordinates \((u,v)\) where \(u = \lfloor \frac{x}{\text{twidth}} \rfloor\), \(v = \lfloor \frac{y}{\text{height}} \rfloor\), and `twidth` and `height` are the width and height of the implementation-dependent shading rate texel size. If the shading rate image view has a type of `VK_IMAGE_VIEW_TYPE_2D_ARRAY`, the lookup will use texel coordinates \((u,v)\) to extract a texel from the layer \(l\), where \(l\) is the layer of the framebuffer being rendered to. If \(l\) is greater than or equal to the number of layers in the image view, layer zero will be used.

If the bound shading rate image view is not `VK_NULL_HANDLE` and contains a texel with coordinates \((u,v)\) in layer \(l\) (if applicable), the single unsigned integer component for that texel will be used as the shading rate index. If the \((u,v)\) coordinate is outside the extents of the subresource used by the shading rate image view, or if the image view is `VK_NULL_HANDLE`, the shading rate index is zero. If the shading rate image view has multiple mipmap levels, the base level identified by `VkImageSubresourceRange::baseMipLevel` will be used.

A shading rate index is mapped to a base shading rate using a lookup table called the shading rate image palette. There is a separate palette for each viewport. The number of entries in each palette is given by the implementation-dependent shading rate image palette size.

If a pipeline state object is created with `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` enabled, the per-viewport shading rate image palettes are set by the command:

```c
// Provided by VK_NV_shading_rate_image
void vkCmdSetViewportShadingRatePaletteNV(
    VkCommandBuffer commandBuffer,
    uint32_t firstViewport,
    uint32_t viewportCount,
    const VkShadingRatePaletteNV* pShadingRatePalettes);
```
• **commandBuffer** is the command buffer into which the command will be recorded.

• **firstViewport** is the index of the first viewport whose shading rate palette is updated by the command.

• **viewportCount** is the number of viewports whose shading rate palettes are updated by the command.

• **pShadingRatePalettes** is a pointer to an array of *VkShadingRatePaletteNV* structures defining the palette for each viewport.

---

**Valid Usage**

- VUID-vkCmdSetViewportShadingRatePaletteNV-None-02064
  
The shading rate image feature must be enabled

- VUID-vkCmdSetViewportShadingRatePaletteNV-firstViewport-02067
  
The sum of firstViewport and viewportCount must be between 1 and *VkPhysicalDeviceLimits::maxViewports*, inclusive

- VUID-vkCmdSetViewportShadingRatePaletteNV-firstViewport-02068
  
  If the multiple viewports feature is not enabled, firstViewport must be 0

- VUID-vkCmdSetViewportShadingRatePaletteNV-viewportCount-02069
  
  If the multiple viewports feature is not enabled, viewportCount must be 1

---

**Valid Usage (Implicit)**

- VUID-vkCmdSetViewportShadingRatePaletteNV-commandBuffer-parameter
  
  commandBuffer must be a valid *VkCommandBuffer* handle

- VUID-vkCmdSetViewportShadingRatePaletteNV-pShadingRatePalettes-parameter
  
  pShadingRatePalettes must be a valid pointer to an array of viewportCount valid *VkShadingRatePaletteNV* structures

- VUID-vkCmdSetViewportShadingRatePaletteNV-commandBuffer-recording
  
  commandBuffer must be in the recording state

- VUID-vkCmdSetViewportShadingRatePaletteNV-commandBuffer-cmdpool
  
  The *VkCommandPool* that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdSetViewportShadingRatePaletteNV-viewportCount-arraylength
  
  viewportCount must be greater than 0

---

**Host Synchronization**

- Host access to commandBuffer must be externally synchronized

- Host access to the *VkCommandPool* that commandBuffer was allocated from must be externally synchronized
The `VkShadingRatePaletteNV` structure specifies the contents of a single shading rate image palette and is defined as:

```c
// Provided by VK_NV_shading_rate_image
typedef struct VkShadingRatePaletteNV {
    uint32_t shadingRatePaletteEntryCount;
    const VkShadingRatePaletteEntryNV* pShadingRatePaletteEntries;
} VkShadingRatePaletteNV;
```

- `shadingRatePaletteEntryCount` specifies the number of entries in the shading rate image palette.
- `pShadingRatePaletteEntries` is a pointer to an array of `VkShadingRatePaletteEntryNV` enums defining the shading rate for each palette entry.

### Valid Usage

- **VUID-VkShadingRatePaletteNV-shadingRatePaletteEntryCount-02071**
  - `shadingRatePaletteEntryCount` must be between `1` and `VkPhysicalDeviceShadingRateImagePropertiesNV::shadingRatePaletteSize`, inclusive.

### Valid Usage (Implicit)

- **VUID-VkShadingRatePaletteNV-pShadingRatePaletteEntries-parameter**
  - `pShadingRatePaletteEntries` must be a valid pointer to an array of `shadingRatePaletteEntryCount` valid `VkShadingRatePaletteEntryNV` values.
- **VUID-VkShadingRatePaletteNV-shadingRatePaletteEntryCount-arraylength**
  - `shadingRatePaletteEntryCount` must be greater than `0`.

To determine the base shading rate image, a shading rate index `i` is mapped to array element `i` in the array `pShadingRatePaletteEntries` for the palette corresponding to the viewport used for the fragment. If `i` is greater than or equal to the palette size `shadingRatePaletteEntryCount`, the base shading rate is undefined.

The supported shading rate image palette entries are defined by `VkShadingRatePaletteEntryNV`:
typedef enum VkShadingRatePaletteEntryNV {
    VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV = 0,
    VK_SHADING_RATE_PALETTE_ENTRY_16_INVOCATIONS_PER_PIXEL_NV = 1,
    VK_SHADING_RATE_PALETTE_ENTRY_8_INVOCATIONS_PER_PIXEL_NV = 2,
    VK_SHADING_RATE_PALETTE_ENTRY_4_INVOCATIONS_PER_PIXEL_NV = 3,
    VK_SHADING_RATE_PALETTE_ENTRY_2_INVOCATIONS_PER_PIXEL_NV = 4,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_PIXEL_NV = 5,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X1_PIXELS_NV = 6,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_1X2_PIXELS_NV = 7,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X2_PIXELS_NV = 8,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X2_PIXELS_NV = 9,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X4_PIXELS_NV = 10,
    VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X4_PIXELS_NV = 11,
} VkShadingRatePaletteEntryNV;

The following table indicates the width and height (in pixels) of each fragment generated using the indicated shading rate, as well as the maximum number of fragment shader invocations launched for each fragment. When processing regions of a primitive that have a shading rate of VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV, no fragments will be generated in that region.

<table>
<thead>
<tr>
<th>Shading Rate</th>
<th>Width</th>
<th>Height</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_NO_INVOCATIONS_NV</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_16_INVOCATIONS_PER_PIXEL_NV</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_8_INVOCATIONS_PER_PIXEL_NV</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_4_INVOCATIONS_PER_PIXEL_NV</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_2_INVOCATIONS_PER_PIXEL_NV</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_PIXEL_NV</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X1_PIXELS_NV</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_1X2_PIXELS_NV</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X2_PIXELS_NV</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Shading Rate</td>
<td>Width</td>
<td>Height</td>
<td>Invocations</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X2_PIXELS_NV</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_2X4_PIXELS_NV</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_4X4_PIXELS_NV</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

When the shading rate image is disabled, a shading rate of `VK_SHADING_RATE_PALETTE_ENTRY_1_INVOCATION_PER_PIXEL_NV` will be used as the base shading rate.

Once a base shading rate has been established, it is adjusted to produce a final shading rate. First, if the base shading rate uses multiple pixels for each fragment, the implementation may reduce the fragment area to ensure that the total number of coverage samples for all pixels in a fragment does not exceed an implementation-dependent maximum.

If sample shading is active in the current pipeline and would result in processing \( n \) \((n > 1)\) unique samples per fragment when the shading rate image is disabled, the shading rate is adjusted in an implementation-dependent manner to increase the number of fragment shader invocations spawned by the primitive. If the shading rate indicates \( f_s \) pixels per fragment and \( f_s \) is greater than \( n \), the fragment area is adjusted so each fragment has approximately \( f_s^\frac{n}{n} \) pixels. Otherwise, if the shading rate indicates \( ipf \) invocations per fragment, the fragment area will be adjusted to a single pixel with approximately \( ipf \times \frac{1}{f_s} \) invocations per fragment.

If sample shading occurs due to the use of a fragment shader input variable decorated with SampleId or SamplePosition, the shading rate is ignored. Each fragment will have a single pixel and will spawn up to `totalSamples` fragment shader invocations, as when using sample shading without a shading rate image.

Finally, if the shading rate specifies multiple fragment shader invocations per fragment, the total number of invocations in the shading rate is clamped to be no larger than the value of `totalSamples` used for sample shading.

When the final shading rate for a primitive covering pixel \((x,y)\) has a fragment area of \( f_w \times f_h \), the fragment for that pixel will cover all pixels with coordinates \( (x',y') \) that satisfy the equations:

\[
\begin{align*}
|\frac{x}{f_w}| &= |\frac{x'}{f_w}| \\
|\frac{y}{f_h}| &= |\frac{y'}{f_h}|
\end{align*}
\]

This combined fragment is considered to have multiple coverage samples; the total number of samples in this fragment is given by \( samples = f_w \times f_h \times rs \) where \( rs \) indicates the value of `VkPipelineMultisampleStateCreateInfo::rasterizationSamples` specified at pipeline creation time. The set of coverage samples in the fragment is the union of the per-pixel coverage samples in each of the fragment's pixels. The location and order of coverage samples within each pixel in the combined
Fragment are assigned as described in Multisampling and Custom Sample Locations. Each coverage sample in the set of pixels belonging to the combined fragment is assigned a unique coverage index in the range \([0, \text{samples}-1]\). If the shadingRateCoarseSampleOrder feature is supported, the order of coverage samples can be specified for each combination of fragment area and coverage sample count. If this feature is not supported, the sample order is implementation-dependent.

If the \(\text{pNext}\) chain of \(\text{VkPipelineViewportStateCreateInfo}\) includes a \(\text{VkPipelineViewportCoarseSampleOrderStateCreateInfoNV}\) structure, then that structure includes parameters that control the order of coverage samples in fragments larger than one pixel.

The \(\text{VkPipelineViewportCoarseSampleOrderStateCreateInfoNV}\) structure is defined as:

```c
typedef struct VkPipelineViewportCoarseSampleOrderStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkCoarseSampleOrderTypeNV sampleOrderType;
    uint32_t customSampleOrderCount;
    const VkCoarseSampleOrderCustomNV* pCustomSampleOrders;
} VkPipelineViewportCoarseSampleOrderStateCreateInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **sampleOrderType** specifies the mechanism used to order coverage samples in fragments larger than one pixel.
- **customSampleOrderCount** specifies the number of custom sample orderings to use when ordering coverage samples.
- **pCustomSampleOrders** is a pointer to an array of customSampleOrderCount \(\text{VkCoarseSampleOrderCustomNV}\) structures, each of which specifies the coverage sample order for a single combination of fragment area and coverage sample count.

If this structure is not present, **sampleOrderType** is considered to be **VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV**.

If **sampleOrderType** is **VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV**, the coverage sample order used for any combination of fragment area and coverage sample count not enumerated in **pCustomSampleOrders** will be identical to that used for **VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV**.

If the pipeline was created with **VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV**, the contents of this structure (if present) are ignored, and the coverage sample order is instead specified by **vkCmdSetCoarseSampleOrderNV**.
Valid Usage

- VUID-VkPipelineViewportCoarseSampleOrderStateCreateInfoNV-sampleOrderType-02072
  If `sampleOrderType` is not `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV`, `customSamplerOrderCount` must be 0

- VUID-VkPipelineViewportCoarseSampleOrderStateCreateInfoNV-pCustomSampleOrders-02234
  The array `pCustomSampleOrders` must not contain two structures with matching values for both the `shadingRate` and `sampleCount` members

Valid Usage (Implicit)

- VUID-VkPipelineViewportCoarseSampleOrderStateCreateInfoNV-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_COARSE_SAMPLE_ORDER_STATE_CREATE_INFO_NV`

- VUID-VkPipelineViewportCoarseSampleOrderStateCreateInfoNV-sampleOrderType-parameter
  `sampleOrderType` must be a valid `VkCoarseSampleOrderTypeNV` value

- VUID-VkPipelineViewportCoarseSampleOrderStateCreateInfoNV-pCustomSampleOrders-parameter
  If `customSampleOrderCount` is not 0, `pCustomSampleOrders` must be a valid pointer to an array of `customSampleOrderCount` valid `VkCoarseSampleOrderCustomNV` structures

The type `VkCoarseSampleOrderTypeNV` specifies the technique used to order coverage samples in fragments larger than one pixel, and is defined as:

```
// Provided by VK_NV_shading_rate_image
typedef enum VkCoarseSampleOrderTypeNV {
    VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV = 0,
    VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV = 1,
    VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV = 2,
    VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV = 3,
} VkCoarseSampleOrderTypeNV;
```

- `VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV` specifies that coverage samples will be ordered in an implementation-dependent manner.
- `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV` specifies that coverage samples will be ordered according to the array of custom orderings provided in either the `pCustomSampleOrders` member of `VkPipelineViewportCoarseSampleOrderStateCreateInfoNV` or the `pCustomSampleOrders` member of `vkCmdSetCoarseSampleOrderNV`.
- `VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV` specifies that coverage samples will be ordered sequentially, sorted first by pixel coordinate (in row-major order) and then by `sample index`.
- `VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV` specifies that coverage samples will be ordered sequentially, sorted first by `sample index` and then by pixel coordinate (in row-major order).

When using a coarse sample order of `VK_COARSE_SAMPLE_ORDER_TYPE_PIXEL_MAJOR_NV` for a fragment
with an upper-left corner of \((f_x, f_y)\) with a width of \(f_w \times f_h\) and \(f_{sc}\) samples per pixel, coverage index \(c_s\) of the fragment will be assigned to sample index \(f_s\) of pixel \((p_x, p_y)\) as follows:

\[
px = f_x + \lfloor \frac{s}{f_{sc}} \rfloor \% f_w \\
py = f_y + \lfloor \frac{s}{f_{sc} \times f_w} \rfloor \\
f_s = cs \% f_{sc}
\]

When using a coarse sample order of `VK_COARSE_SAMPLE_ORDER_TYPE_SAMPLE_MAJOR_NV`, coverage index \(c_s\) will be assigned as follows:

\[
px = f_x + cs \% f_w \\
py = (f_y + \lfloor \frac{s}{f_{sc}} \rfloor \% f_h) \\
f_s = \lfloor \frac{s}{f_w \times f_h} \rfloor
\]

The `VkCoarseSampleOrderCustomNV` structure is used with a coverage sample ordering type of `VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV` to specify the order of coverage samples for one combination of fragment width, fragment height, and coverage sample count. The structure is defined as:

```c
// Provided by VK_NV_shading_rate_image
typedef struct VkCoarseSampleOrderCustomNV {
    VkShadingRatePaletteEntryNV shadingRate;
    uint32_t sampleCount;
    uint32_t sampleLocationCount;
    const VkCoarseSampleLocationNV* pSampleLocations;
} VkCoarseSampleOrderCustomNV;
```

- `shadingRate` is a shading rate palette entry that identifies the fragment width and height for the combination of fragment area and per-pixel coverage sample count to control.
- `sampleCount` identifies the per-pixel coverage sample count for the combination of fragment area and coverage sample count to control.
- `sampleLocationCount` specifies the number of sample locations in the custom ordering.
- `pSampleLocations` is a pointer to an array of `VkCoarseSampleLocationNV` structures specifying the location of each sample in the custom ordering.

When using a custom sample ordering, element \(j\) in `pSampleLocations` specifies a specific pixel location and sample index that corresponds to coverage index \(j\) in the multi-pixel fragment.
Valid Usage

- VUID-VkCoarseSampleOrderCustomNV-shadingRate-02073
  
  **shadingRate** must be a shading rate that generates fragments with more than one pixel

- VUID-VkCoarseSampleOrderCustomNV-sampleCount-02074
  
  **sampleCount** must correspond to a sample count enumerated in VkSampleCountFlags whose corresponding bit is set in VkPhysicalDeviceLimits::framebufferNoAttachmentsSampleCounts

- VUID-VkCoarseSampleOrderCustomNV-sampleLocationCount-02075
  
  **sampleLocationCount** must be equal to the product of **sampleCount**, the fragment width for shadingRate, and the fragment height for shadingRate

- VUID-VkCoarseSampleOrderCustomNV-sampleLocationCount-02076
  
  **sampleLocationCount** must be less than or equal to the value of VkPhysicalDeviceShadingRateImagePropertiesNV::shadingRateMaxCoarseSamples

- VUID-VkCoarseSampleOrderCustomNV-pSampleLocations-02077
  
  The array **pSampleLocations** must contain exactly one entry for every combination of valid values for pixelX, pixelY, and sample in the structure VkCoarseSampleOrderCustomNV

Valid Usage (Implicit)

- VUID-VkCoarseSampleOrderCustomNV-shadingRate-parameter
  
  **shadingRate** must be a valid VkShadingRatePaletteEntryNV value

- VUID-VkCoarseSampleOrderCustomNV-pSampleLocations-parameter
  
  **pSampleLocations** must be a valid pointer to an array of sampleLocationCount VkCoarseSampleLocationNV structures

- VUID-VkCoarseSampleOrderCustomNV-sampleLocationCount-arraylength
  
  **sampleLocationCount** must be greater than 0

The VkCoarseSampleLocationNV structure identifies a specific pixel and sample index for one of the coverage samples in a fragment that is larger than one pixel. This structure is defined as:

```c
// Provided by VK_NV_shading_rate_image
typedef struct VkCoarseSampleLocationNV {
    uint32_t    pixelX;
    uint32_t    pixelY;
    uint32_t    sample;
} VkCoarseSampleLocationNV;
```

- **pixelX** is added to the x coordinate of the upper-leftmost pixel of each fragment to identify the pixel containing the coverage sample.
- **pixelY** is added to the y coordinate of the upper-leftmost pixel of each fragment to identify the pixel containing the coverage sample.
• \textit{sample} is the number of the coverage sample in the pixel identified by \textit{pixelX} and \textit{pixelY}.

\textbf{Valid Usage}

• \textbf{VUID-VkCoarseSampleLocationNV-pixelX-02078}
  \textit{pixelX} \textbf{must} be less than the width (in pixels) of the fragment

• \textbf{VUID-VkCoarseSampleLocationNV-pixelY-02079}
  \textit{pixelY} \textbf{must} be less than the height (in pixels) of the fragment

• \textbf{VUID-VkCoarseSampleLocationNV-sample-02080}
  \textit{sample} \textbf{must} be less than the number of coverage samples in each pixel belonging to the fragment

If a pipeline state object is created with \texttt{VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV} enabled, the order of coverage samples in fragments larger than one pixel is set by the command:

```c
// Provided by VK_NV_shading_rate_image
void vkCmdSetCoarseSampleOrderNV(
    VkCommandBuffer                             commandBuffer,
    VkCoarseSampleOrderTypeNV                   sampleOrderType,
    uint32_t                                    customSampleOrderCount,
    const VkCoarseSampleOrderCustomNV*          pCustomSampleOrders);
```

• \textit{commandBuffer} is the command buffer into which the command will be recorded.

• \textit{sampleOrderType} specifies the mechanism used to order coverage samples in fragments larger than one pixel.

• \textit{customSampleOrderCount} specifies the number of custom sample orderings to use when ordering coverage samples.

• \textit{pCustomSampleOrders} is a pointer to an array of \texttt{VkCoarseSampleOrderCustomNV} structures, each of which specifies the coverage sample order for a single combination of fragment area and coverage sample count.

If \textit{sampleOrderType} is \texttt{VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV}, the coverage sample order used for any combination of fragment area and coverage sample count not enumerated in \textit{pCustomSampleOrders} will be identical to that used for \texttt{VK_COARSE_SAMPLE_ORDER_TYPE_DEFAULT_NV}.

\textbf{Valid Usage}

• \textbf{VUID-vkCmdSetCoarseSampleOrderNV-sampleOrderType-02081}
  \textbf{If} \textit{sampleOrderType} \textbf{is not} \texttt{VK_COARSE_SAMPLE_ORDER_TYPE_CUSTOM_NV}, \textit{customSamplerOrderCount} \textbf{must} be 0

• \textbf{VUID-vkCmdSetCoarseSampleOrderNV-pCustomSampleOrders-02235}
  The array \textit{pCustomSampleOrders} \textbf{must} not contain two structures with matching values for both the \texttt{shadingRate} and \texttt{sampleCount} members
Valid Usage (Implicit)

- VUID-vkCmdSetCoarseSampleOrderNV-commandBuffer-parameter
  \texttt{commandBuffer} \textbf{must} be a valid \texttt{VkCommandBuffer} handle

- VUID-vkCmdSetCoarseSampleOrderNV-sampleOrderType-parameter
  \texttt{sampleOrderType} \textbf{must} be a valid \texttt{VkCoarseSampleOrderTypeNV} value

- VUID-vkCmdSetCoarseSampleOrderNV-pCustomSampleOrders-parameter
  If \texttt{customSampleOrderCount} is not 0, \texttt{pCustomSampleOrders} \textbf{must} be a valid pointer to an array of \texttt{customSampleOrderCount} valid \texttt{VkCoarseSampleOrderCustomNV} structures

- VUID-vkCmdSetCoarseSampleOrderNV-commandBuffer-recording
  \texttt{commandBuffer} \textbf{must} be in the \textit{recording state}

- VUID-vkCmdSetCoarseSampleOrderNV-commandBuffer-cmdpool
  The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from \textbf{must} support graphics operations

Host Synchronization

- Host access to \texttt{commandBuffer} \textbf{must} be externally synchronized

- Host access to the \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from \textbf{must} be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the final shading rate for a primitive covering pixel \((x, y)\) results in \(n\) invocations per pixel \((n > 1)\), \(n\) separate fragment shader invocations will be generated for the fragment. Each coverage sample in the fragment will be assigned to one of the \(n\) fragment shader invocations in an implementation-dependent manner. The outputs from the \textit{fragment output interface} of each shader invocation will be broadcast to all of the framebuffer samples associated with the invocation. If none of the coverage samples associated with a fragment shader invocation is covered by a primitive, the implementation \textbf{may} discard the fragment shader invocation for those samples.

If the final shading rate for a primitive covering pixel \((x, y)\) results in a fragment containing multiple pixels, a single set of fragment shader invocations will be generated for all pixels in the combined fragment. Outputs from the \textit{fragment output interface} will be broadcast to all covered framebuffer samples belonging to the fragment. If the fragment shader executes code discarding the fragment, none of the samples of the fragment will be updated.
27.8. Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. If sample shading is enabled an implementation must provide a minimum of \( \text{max}(\left\lceil \text{minSampleShadingFactor} \times \text{totalSamples} \right\rceil, 1) \) unique associated data for each fragment, where \( \text{minSampleShadingFactor} \) is the minimum fraction of sample shading. If the \text{VK_AMD_mixed_attachment_samples} extension is enabled and the subpass uses color attachments, \( \text{totalSamples} \) is the number of samples of the color attachments. Otherwise, \( \text{totalSamples} \) is the value of \text{VkPipelineMultisampleStateCreateInfo::rasterizationSamples} specified at pipeline creation time. These are associated with the samples in an implementation-dependent manner. When \( \text{minSampleShadingFactor} \) is 1.0, a separate set of associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

Sample shading is enabled for a graphics pipeline:

- If the interface of the fragment shader entry point of the graphics pipeline includes an input variable decorated with \text{SampleId} or \text{SamplePosition}. In this case \( \text{minSampleShadingFactor} \) takes the value 1.0.
- Else if the \text{sampleShadingEnable} member of the \text{VkPipelineMultisampleStateCreateInfo} structure specified when creating the graphics pipeline is set to \text{VK_TRUE}. In this case \( \text{minSampleShadingFactor} \) takes the value of \text{VkPipelineMultisampleStateCreateInfo::minSampleShading}.

Otherwise, sample shading is considered disabled.

27.9. Barycentric Interpolation

When the \text{fragmentShaderBarycentric} feature is enabled, the \text{PerVertexNV} interpolation decoration can be used with fragment shader inputs to indicate that the decorated inputs do not have associated data in the fragment. Such inputs can only be accessed in a fragment shader using an array index whose value (0, 1, or 2) identifies one of the vertices of the primitive that produced the fragment.

When tessellation, geometry shading, and mesh shading are not active, fragment shader inputs decorated with \text{PerVertexNV} will take values from one of the vertices of the primitive that produced the fragment, identified by the extra index provided in SPIR-V code accessing the input. If the \( n \) vertices passed to a draw call are numbered 0 through \( n-1 \), and the point, line, and triangle primitives produced by the draw call are numbered with consecutive integers beginning with zero, the following table indicates the original vertex numbers used when the provoking vertex mode is \text{VK_PROVOKING_VERTEX_MODE_FIRST_VERTEX_EXT} for index values of 0, 1, and 2. If an input decorated with \text{PerVertexNV} is accessed with any other vertex index value, an undefined value is returned.

<table>
<thead>
<tr>
<th>Primitive Topology</th>
<th>Vertex 0</th>
<th>Vertex 1</th>
<th>Vertex 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{VK_PRIMITIVE_TOPOLOGY_POINT_LIST}</td>
<td>i</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>\text{VK_PRIMITIVE_TOPOLOGY_LINE_LIST}</td>
<td>2i</td>
<td>2i+1</td>
<td>-</td>
</tr>
</tbody>
</table>
When the provoking vertex mode is `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT`, the original vertex numbers used are the same as above except as indicated in the table below.

<table>
<thead>
<tr>
<th>Primitive Topology</th>
<th>Vertex 0</th>
<th>Vertex 1</th>
<th>Vertex 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_LINE_STRIP</code></td>
<td>i</td>
<td>i+1</td>
<td>-</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST</code></td>
<td>3i</td>
<td>3i+1</td>
<td>3i+2</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP</code> (even)</td>
<td>i</td>
<td>i+1</td>
<td>i+2</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP</code> (odd)</td>
<td>i</td>
<td>i+2</td>
<td>i+1</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN</code></td>
<td>i+1</td>
<td>i+2</td>
<td>0</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY</code></td>
<td>4i+1</td>
<td>4i+2</td>
<td>-</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY</code></td>
<td>i+1</td>
<td>i+2</td>
<td>-</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY</code></td>
<td>6i</td>
<td>6i+2</td>
<td>6i+4</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</code> (even)</td>
<td>2i</td>
<td>2i+2</td>
<td>2i+4</td>
</tr>
<tr>
<td><code>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY</code> (odd)</td>
<td>2i</td>
<td>2i+4</td>
<td>2i+2</td>
</tr>
</tbody>
</table>

When geometry or mesh shading is active, primitives processed by fragment shaders are assembled from the vertices emitted by the geometry or mesh shader. In this case, the vertices used for fragment shader inputs decorated with `PerVertexNV` are derived by treating the primitives produced by the shader as though they were specified by a draw call and consulting the table above.

When using tessellation without geometry shading, the tessellator produces primitives in an implementation-dependent manner. While there is no defined vertex ordering for inputs decorated with `PerVertexNV`, the vertex ordering used in this case will be consistent with the ordering used to
derive the values of inputs decorated with \texttt{BaryCoordNV} or \texttt{BaryCoordNoPerspNV}.

Fragment shader inputs decorated with \texttt{BaryCoordNV} or \texttt{BaryCoordNoPerspNV} hold three-component vectors with barycentric weights that indicate the location of the fragment relative to the screen-space locations of vertices of its primitive. For point primitives, such variables are always assigned the value (1,0,0). For line primitives, the built-ins are obtained by interpolating an attribute whose values for the vertices numbered 0 and 1 are (1,0,0) and (0,1,0), respectively. For polygon primitives, the built-ins are obtained by interpolating an attribute whose values for the vertices numbered 0, 1, and 2 are (1,0,0), (0,1,0), and (0,0,1), respectively. For \texttt{BaryCoordNV}, the values are obtained using perspective interpolation. For \texttt{BaryCoordNoPerspNV}, the values are obtained using linear interpolation.

### 27.10. Points

A point is drawn by generating a set of fragments in the shape of a square centered around the vertex of the point. Each vertex has an associated point size that controls the width/height of that square. The point size is taken from the (potentially clipped) shader built-in \texttt{PointSize} written by:

- the geometry shader, if active;
- the tessellation evaluation shader, if active and no geometry shader is active;
- the vertex shader, otherwise

and clamped to the implementation-dependent point size range \([\text{pointSizeRange}[0], \text{pointSizeRange}[1]]\). The value written to \texttt{PointSize} must be greater than zero.

Not all point sizes need be supported, but the size 1.0 must be supported. The range of supported sizes and the size of evenly-spaced gradations within that range are implementation-dependent. The range and gradations are obtained from the \texttt{pointSizeRange} and \texttt{pointSizeGranularity} members of \texttt{VkPhysicalDeviceLimits}. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the sizes 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional point sizes may also be supported. There is no requirement that these sizes be equally spaced. If an unsupported size is requested, the nearest supported size is used instead.

Further, if the render pass has a fragment density map attachment, point size may be rounded by the implementation to a multiple of the fragment's width or height.

### 27.10.1. Basic Point Rasterization

Point rasterization produces a fragment for each fragment area group of framebuffer pixels with one or more sample points that intersect a region centered at the point's \((x_f,y_f)\). This region is a square with side equal to the current point size. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All fragments produced in rasterizing a point are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader built-in \texttt{PointCoord} contains point sprite texture coordinates. The s and t point sprite texture coordinates vary from zero to one across the point horizontally left-to-right and vertically top-to-bottom, respectively. The following formulas are used to evaluate s and t:

\[
s = \frac{1}{2} + \frac{(x_f - x)}{\text{size}}
\]
where \( t = \frac{1}{2} + \frac{(y_p - y_f)}{\text{size}} \)

The location at which the point sprite coordinates are evaluated - this may be the framebuffer coordinates of the fragment center, or the location of a sample; and \((x_i, y_i)\) is the exact, unrounded framebuffer coordinate of the vertex for the point.

27.11. Line Segments

Line segment rasterization options are controlled by the `VkPipelineRasterizationLineStateCreateInfoEXT` structure.

The `VkPipelineRasterizationLineStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_line_rasterization
typedef struct VkPipelineRasterizationLineStateCreateInfoEXT {
    VkStructureType               sType;
    const void*                   pNext;
    VkLineRasterizationModeEXT    lineRasterizationMode;
    VkBool32                      stippledLineEnable;
    uint32_t                      lineStippleFactor;
    uint16_t                      lineStipplePattern;
} VkPipelineRasterizationLineStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `lineRasterizationMode` is a `VkLineRasterizationModeEXT` value selecting the style of line rasterization.
- `stippledLineEnable` enables stippled line rasterization.
- `lineStippleFactor` is the repeat factor used in stippled line rasterization.
- `lineStipplePattern` is the bit pattern used in stippled line rasterization.
Valid Usage

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02768
  If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT`, then the `rectangularLines` feature must be enabled

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02769
  If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT`, then the `bresenhamLines` feature must be enabled

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-02770
  If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then the `smoothLines` feature must be enabled

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02771
  If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT`, then the `stippledRectangularLines` feature must be enabled

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02772
  If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT`, then the `stippledBresenhamLines` feature must be enabled

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02773
  If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then the `stippledSmoothLines` feature must be enabled

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-stippledLineEnable-02774
  If `stippledLineEnable` is `VK_TRUE` and `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT`, then the `stippledRectangularLines` feature must be enabled and `VkPhysicalDeviceLimits::strictLines` must be `VK_TRUE`

Valid Usage (Implicit)

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT`

- VUID-VkPipelineRasterizationLineStateCreateInfoEXT-lineRasterizationMode-parameter
  `lineRasterizationMode` must be a valid `VkLineRasterizationModeEXT` value

Possible values of `VkPipelineRasterizationLineStateCreateInfoEXT::lineRasterizationMode` are:
```c
// Provided by VK_EXT_line_rasterization
typedef enum VkLineRasterizationModeEXT {
    VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT = 0,
    VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT = 1,
    VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT = 2,
    VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT = 3,
} VkLineRasterizationModeEXT;
```

- **VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT** is equivalent to **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT** if VkPhysicalDeviceLimits::strictLines is VK_TRUE, otherwise lines are drawn as non-strictLines parallelograms. Both of these modes are defined in Basic Line Segment Rasterization.

- **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT** specifies lines drawn as if they were rectangles extruded from the line.

- **VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT** specifies lines drawn by determining which pixel diamonds the line intersects and exits, as defined in Bresenham Line Segment Rasterization.

- **VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT** specifies lines drawn if they were rectangles extruded from the line, with alpha falloff, as defined in Smooth Lines.

Each line segment has an associated width. The line width is specified by the VkPipelineRasterizationStateCreateInfo::lineWidth property of the currently active pipeline, if the pipeline was not created with VK_DYNAMIC_STATE_LINE_WIDTH enabled.

Otherwise, the line width is set by calling `vkCmdSetLineWidth`:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetLineWidth(
    VkCommandBuffer                             commandBuffer,
    float                                       lineWidth);
```

- **commandBuffer** is the command buffer into which the command will be recorded.

- **lineWidth** is the width of rasterized line segments.

**Valid Usage**

- VUID-vkCmdSetLineWidth-lineWidth-00788

  If the wide lines feature is not enabled, lineWidth must be 1.0
Valid Usage (Implicit)

- **VUID-vkCmdSetLineWidth-commandBuffer-parameter**
  commandBuffer must be a valid VkCommandBuffer handle

- **VUID-vkCmdSetLineWidth-commandBuffer-recording**
  commandBuffer must be in the recording state

- **VUID-vkCmdSetLineWidth-commandBuffer-cmdpool**
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not all line widths need be supported for line segment rasterization, but width 1.0 antialiased segments must be provided. The range and gradations are obtained from the lineWidthRange and lineWidthGranularity members of VkPhysicalDeviceLimits. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the sizes 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional line widths may also be supported. There is no requirement that these widths be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

Further, if the render pass has a fragment density map attachment, line width may be rounded by the implementation to a multiple of the fragment’s width or height.

**27.11.1. Basic Line Segment Rasterization**

If the lineRasterizationMode member of VkPipelineRasterizationLineStateCreateInfoEXT is VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT, rasterized line segments produce fragments which intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment in directions perpendicular to the direction of the line. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage bits that correspond to sample points that intersect the rectangle are 1, other coverage bits are 0.
Next we specify how the data associated with each rasterized fragment are obtained. Let \( p_r = (x_d, y_d) \) be the framebuffer coordinates at which associated data are evaluated. This may be the center of a fragment or the location of a sample within the fragment. When \( \text{rasterizationSamples} \) is \( \text{VK\_SAMPLE\_COUNT\_1\_BIT} \), the fragment center must be used. Let \( p_a = (x_a, y_a) \) and \( p_b = (x_b, y_b) \) be initial and final endpoints of the line segment, respectively. Set

\[
t = \frac{(p_r - p_a) \cdot (p_a - p_b)}{||p_b - p_a||^2}
\]

(Note that \( t = 0 \) at \( p_a \) and \( t = 1 \) at \( p_b \). Also note that this calculation projects the vector from \( p_a \) to \( p_r \) onto the line, and thus computes the normalized distance of the fragment along the line.)

The value of an associated datum \( f \) for the fragment, whether it be a shader output or the clip \( w \) coordinate, must be determined using perspective interpolation:

\[
f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b}
\]

where \( f_a \) and \( f_b \) are the data associated with the starting and ending endpoints of the segment, respectively; \( w_a \) and \( w_b \) are the clip \( w \) coordinates of the starting and ending endpoints of the segment, respectively.

Depth values for lines must be determined using linear interpolation:

\[
z = (1-t)z_a + tz_b
\]

where \( z_a \) and \( z_b \) are the depth values of the starting and ending endpoints of the segment, respectively.

The \texttt{NoPerspective} and \texttt{Flat} interpolation decorations can be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, perspective interpolation is performed as described above. When the \texttt{NoPerspective} decoration is used, linear interpolation is performed in the same fashion as for depth values, as described above. When the \texttt{Flat} decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive.

When the \texttt{fragmentShaderBarycentric} feature is enabled, the \texttt{PerVertexNV} interpolation decoration can also be used with fragment shader inputs which indicate that the decorated inputs are not interpolated and can only be accessed using an extra array dimension, where the extra index identifies one of the vertices of the primitive that produced the fragment.

The above description documents the preferred method of line rasterization, and must be used when the implementation advertises the \texttt{strictLines} limit in \texttt{VkPhysicalDeviceLimits} as \texttt{VK\_TRUE}.

When \texttt{strictLines} is \texttt{VK\_FALSE}, the edges of the lines are generated as a parallelogram surrounding the original line. The major axis is chosen by noting the axis in which there is the greatest distance between the line start and end points. If the difference is equal in both directions then the X axis is chosen as the major axis. Edges 2 and 3 are aligned to the minor axis and are centered on the endpoints of the line as in \texttt{NonStrictLines}, and each is \texttt{lineWidth} long. Edges 0 and 1 are parallel to the line and connect the endpoints of edges 2 and 3. Coverage bits that correspond to sample points
that intersect the parallelogram are 1, other coverage bits are 0.

Samples that fall exactly on the edge of the parallelogram follow the polygon rasterization rules.

Interpolation occurs as if the parallelogram was decomposed into two triangles where each pair of vertices at each end of the line has identical attributes.

![Diagram of a parallelogram with labeled edges and a line segment](image)

Figure 17. Non strict lines

Only when `strictLines` is `VK_FALSE` implementations **may** deviate from the non-strict line algorithm described above in the following ways:

- Implementations **may** instead interpolate each fragment according to the formula in [Basic Line Segment Rasterization](#) using the original line segment endpoints.
- Rasterization of non-antialiased non-strict line segments **may** be performed using the rules defined in [Bresenham Line Segment Rasterization](#).

### 27.11.2. Bresenham Line Segment Rasterization

If `lineRasterizationMode` is `VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT`, then the following rules replace the line rasterization rules defined in [Basic Line Segment Rasterization](#).

Non-strict lines **may** also follow these rasterization rules for non-antialiased lines.

Line segment rasterization begins by characterizing the segment as either **x-major** or **y-major**. x-major line segments have slope in the closed interval [-1,1]; all other line segments are y-major (slope is determined by the segment's endpoints). We specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.

Ideally, Vulkan uses a **diamond-exit** rule to determine those fragments that are produced by rasterizing a line segment. For each fragment $f$ with center at framebuffer coordinates $x_f$ and $y_f$, define a diamond-shaped region that is the intersection of four half planes:

$$ R_f = \{(x, y) | |x - x_f| + |y - y_f| < \frac{1}{2}\} $$

Essentially, a line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment intersects $R_c$, except if $p_b$ is contained in $R_c$. 
To avoid difficulties when an endpoint lies on a boundary of $R_f$ we (in principle) perturb the supplied endpoints by a tiny amount. Let $p_a$ and $p_b$ have framebuffer coordinates $(x_a, y_a)$ and $(x_b, y_b)$, respectively. Obtain the perturbed endpoints $p_a'$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $p_b'$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment starting at $p_a'$ and ending on $p_b'$ intersects $R_f$, except if $p_b'$ is contained in $R_f$. $\epsilon$ is chosen to be so small that rasterizing the line segment produces the same fragments when $\delta$ is substituted for $\epsilon$ for any $0 < \delta \leq \epsilon$.

When $p_a$ and $p_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham’s algorithm with one modification: lines produced in this description are “half-open,” meaning that the final fragment (corresponding to $p_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham’s algorithm).

Implementations may use other line segment rasterization algorithms, subject to the following rules:

- The coordinates of a fragment produced by the algorithm must not deviate by more than one unit in either x or y framebuffer coordinates from a corresponding fragment produced by the diamond-exit rule.
- The total number of fragments produced by the algorithm must not differ from that produced by the diamond-exit rule by no more than one.
- For an x-major line, two fragments that lie in the same framebuffer-coordinate column must not be produced (for a y-major line, two fragments that lie in the same framebuffer-coordinate row must not be produced).
- If two line segments share a common endpoint, and both segments are either x-major (both left-to-right or both right-to-left) or y-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments must not produce duplicate fragments. Fragments also must not be omitted so as to interrupt continuity of the connected segments.
The actual width \( w \) of Bresenham lines is determined by rounding the line width to the nearest integer, clamping it to the implementation-dependent \( \text{lineWidthRange} \) (with both values rounded to the nearest integer), then clamping it to be no less than 1.

Bresenham line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y, and for a y-major line, the minor direction is x) and producing a row or column of fragments in the minor direction. If the line segment has endpoints given by \((x_0, y_0)\) and \((x_1, y_1)\) in framebuffer coordinates, the segment with endpoints \((x_0, y_0 - \frac{w-1}{2})\) and \((x_1, y_1 - \frac{w-1}{2})\) is rasterized, but instead of a single fragment, a column of fragments of height \( w \) (a row of fragments of length \( w \) for a y-major segment) is produced at each x (y for y-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates.

The preferred method of attribute interpolation for a wide line is to generate the same attribute values for all fragments in the row or column described above, as if the adjusted line was used for interpolation and those values replicated to the other fragments, except for \( \text{FragCoord} \) which is interpolated as usual. Implementations may instead interpolate each fragment according to the formula in Basic Line Segment Rasterization, using the original line segment endpoints.

When Bresenham lines are being rasterized, sample locations may all be treated as being at the pixel center (this may affect attribute and depth interpolation).

\[ b = \left\lfloor \frac{s}{r} \right\rfloor \mod 16 \]

Then a fragment is produced if the \( b \)th bit of \( p \) is 1, and discarded otherwise. The bits of \( p \) are numbered with 0 being the least significant and 15 being the most significant.
The initial value of $s$ is zero. For \texttt{VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT} lines, $s$ is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). For \texttt{VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT} and \texttt{VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT} lines, the rectangular region is subdivided into adjacent unit-length rectangles, and $s$ is incremented once for each rectangle. Rectangles with a value of $s$ such that the $b$'th bit of $p$ is zero are discarded. If the last rectangle in a line segment is shorter than unit-length, then the remainder \textbf{may} carry over to the next line segment in the line strip using the same value of $s$ (this is the preferred behavior, for the stipple pattern to appear more consistent through the strip).

$s$ is reset to 0 at the start of each strip (for line strips), and before every line segment in a group of independent segments.

If the line segment has been clipped, then the value of $s$ at the beginning of the line segment is implementation-dependent.

The line stipple factor and pattern are specified by the \texttt{VkPipelineRasterizationLineStateCreateInfoEXT::lineStippleFactor} and \texttt{VkPipelineRasterizationLineStateCreateInfoEXT::lineStipplePattern} members of the currently active pipeline, if the pipeline was not created with \texttt{VK_DYNAMIC_STATE_LINE_STIPPLE_EXT} enabled.

Otherwise, the line stipple factor and pattern are set by calling \texttt{vkCmdSetLineStippleEXT}:

```c
// Provided by VK_EXT_line_rasterization
void vkCmdSetLineStippleEXT(
    VkCommandBuffer commandBuffer,
    uint32_t lineStippleFactor,
    uint16_t lineStipplePattern);
```

- \texttt{commandBuffer} is the command buffer into which the command will be recorded.
- \texttt{lineStippleFactor} is the repeat factor used in stippled line rasterization.
- \texttt{lineStipplePattern} is the bit pattern used in stippled line rasterization.

**Valid Usage**

- \textbf{VUID-vkCmdSetLineStippleEXT-lineStippleFactor-02776} \texttt{lineStippleFactor} \textbf{must} be in the range $[1,256]$
Valid Usage (Implicit)

- VUID-vkCmdSetLineStippleEXT-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetLineStippleEXT-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetLineStippleEXT-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

27.11.4. Smooth Lines

If the `lineRasterizationMode` member of `VkPipelineRasterizationLineStateCreateInfoEXT` is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, then lines are considered to be rectangles using the same geometry as for `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT` lines. The rules for determining which pixels are covered are implementation-dependent, and may include nearby pixels where no sample locations are covered or where the rectangle doesn't intersect the pixel at all. For each pixel that is considered covered, the fragment computes a coverage value that approximates the area of the intersection of the rectangle with the pixel square, and this coverage value is multiplied into the color location 0’s alpha value after fragment shading, as described in Multisample Coverage.

Note

The details of the rasterization rules and area calculation are left intentionally vague, to allow implementations to generate coverage and values that are aesthetically pleasing.
27.12. Polygons

A polygon results from the decomposition of a triangle strip, triangle fan or a series of independent triangles. Like points and line segments, polygon rasterization is controlled by several variables in the VkPipelineRasterizationStateCreateInfo structure.

27.12.1. Basic Polygon Rasterization

The first step of polygon rasterization is to determine whether the triangle is back-facing or front-facing. This determination is made based on the sign of the (clipped or unclipped) polygon’s area computed in framebuffer coordinates. One way to compute this area is:

\[ a = -\frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i \]

where \( x_i \) and \( y_i \) are the x and y framebuffer coordinates of the ith vertex of the n-vertex polygon (vertices are numbered starting at zero for the purposes of this computation) and \( i \oplus 1 \) is \((i + 1) \mod n\).

The interpretation of the sign of \( a \) is determined by the VkPipelineRasterizationStateCreateInfo::frontFace property of the currently active pipeline. Possible values are:

- **VK_FRONT_FACE_COUNTER_CLOCKWISE** specifies that a triangle with positive area is considered front-facing.
- **VK_FRONT_FACE_CLOCKWISE** specifies that a triangle with negative area is considered front-facing.

Any triangle which is not front-facing is back-facing, including zero-area triangles.

If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_FRONT_FACE_EXT dynamic state enabled then the front face property is set dynamically by calling:

```
// Provided by VK_VERSION_1_0
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **frontFace** specifies the front face property to use for drawing.
Valid Usage

- VUID-vkCmdSetFrontFaceEXT-None-03383
  The `extendedDynamicState` feature must be enabled

Valid Usage (Implicit)

- VUID-vkCmdSetFrontFaceEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetFrontFaceEXT-frontFace-parameter
  `frontFace` must be a valid `VkFrontFace` value

- VUID-vkCmdSetFrontFaceEXT-commandBuffer-recording
  `commandBuffer` must be in the `recording state`

- VUID-vkCmdSetFrontFaceEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary, Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>

Once the orientation of triangles is determined, they are culled according to the `VkPipelineRasterizationStateCreateInfo::cullMode` property of the currently active pipeline. Possible values are:
typedef enum VkCullModeFlagBits {
    VK_CULL_MODE_NONE = 0,
    VK_CULL_MODE_FRONT_BIT = 0x00000001,
    VK_CULL_MODE_BACK_BIT = 0x00000002,
    VK_CULL_MODE_FRONT_AND_BACK = 0x00000003,
} VkCullModeFlagBits;

- **VK_CULL_MODE_NONE** specifies that no triangles are discarded
- **VK_CULL_MODE_FRONT_BIT** specifies that front-facing triangles are discarded
- **VK_CULL_MODE_BACK_BIT** specifies that back-facing triangles are discarded
- **VK_CULL_MODE_FRONT_AND_BACK** specifies that all triangles are discarded.

Following culling, fragments are produced for any triangles which have not been discarded.

typedef VkFlags VkCullModeFlags;

**VkCullModeFlags** is a bitmask type for setting a mask of zero or more **VkCullModeFlagBits**.

If the bound graphics pipeline state was created with the **VK_DYNAMIC_STATE_CULL_MODE_EXT** dynamic state enabled then the cull mode is set dynamically by calling:

```
void vkCmdSetCullModeEXT(
    VkCommandBuffer commandBuffer,
    VkCullModeFlags cullMode);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **cullMode** specifies the cull mode property to use for drawing.

**Valid Usage**

- **VUID-vkCmdSetCullModeEXT-None-03384**
  
  The **extendedDynamicState** feature must be enabled
Valid Usage (Implicit)

- **VUID-vkCmdSetCullModeEXT-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle
- **VUID-vkCmdSetCullModeEXT-cullMode-parameter**
  - `cullMode` must be a valid combination of `VkCullModeFlagBits` values
- **VUID-vkCmdSetCullModeEXT-commandBuffer-recording**
  - `commandBuffer` must be in the recording state
- **VUID-vkCmdSetCullModeEXT-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y framebuffer coordinates of the polygon’s vertices is formed. Fragments are produced for any fragment area groups of pixels for which any sample points lie inside of this polygon. Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Special treatment is given to a sample whose sample location lies on a polygon edge. In such a case, if two polygons lie on either side of a common edge (with identical endpoints) on which a sample point lies, then exactly one of the polygons must result in a covered sample for that fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle.

*Barycentric coordinates* are a set of three numbers, a, b, and c, each in the range [0,1], with $a + b + c = 1$. These coordinates uniquely specify any point $p$ within the triangle or on the triangle’s boundary as

$$p = a p_a + b p_b + c p_c$$
where \( p_a, p_b \), and \( p_c \) are the vertices of the triangle. \( a, b, \) and \( c \) are determined by:

\[
a = \frac{A(p_a p_b p_c) - A(p_a p_b p_c)}{A(p_a p_b p_c)}, \quad b = \frac{A(p_a p_b p_c) - A(p_a p_b p_c)}{A(p_a p_b p_c)}, \quad c = \frac{A(p_a p_b p_c) - A(p_a p_b p_c)}{A(p_a p_b p_c)},
\]

where \( A(lmn) \) denotes the area in framebuffer coordinates of the triangle with vertices \( l, m, \) and \( n \).

Denote an associated datum at \( p_a, p_b, \) or \( p_c \) as \( f_a, f_b, \) or \( f_c \), respectively.

The value of an associated datum \( f \) for a fragment produced by rasterizing a triangle, whether it be a shader output or the clip \( w \) coordinate, must be determined using perspective interpolation:

\[
f = \frac{a f_a / w_a + b f_b / w_b + c f_c / w_c}{a / w_a + b / w_b + c / w_c}
\]

where \( w_a, w_b, \) and \( w_c \) are the clip \( w \) coordinates of \( p_a, p_b, \) and \( p_c \), respectively. \( a, b, \) and \( c \) are the barycentric coordinates of the location at which the data are produced - this must be the location of the fragment center or the location of a sample. When \( \text{rasterizationSamples} \) is \( VK\_SAMPLE\_COUNT\_1\_BIT \), the fragment center must be used.

Depth values for triangles must be determined using linear interpolation:

\[
z = a z_a + b z_b + c z_c
\]

where \( z_a, z_b, \) and \( z_c \) are the depth values of \( p_a, p_b, \) and \( p_c, \) respectively.

The \text{NoPerspective} and \text{Flat} interpolation decorations can be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, perspective interpolation is performed as described above. When the \text{NoPerspective} decoration is used, linear interpolation is performed in the same fashion as for depth values, as described above. When the \text{Flat} decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive.

When the \text{VK_AMD_shader_explicit_vertex_parameter} device extension is enabled the \text{CustomInterpAMD} interpolation decoration can also be used with fragment shader inputs which indicate that the decorated inputs can only be accessed by the extended instruction \text{InterpolateAtVertexAMD} and allows accessing the value of the inputs for individual vertices of the primitive.

When the \text{fragmentShaderBarycentric} feature is enabled, the \text{PerVertexNV} interpolation decoration can also be used with fragment shader inputs which indicate that the decorated inputs are not interpolated and can only be accessed using an extra array dimension, where the extra index identifies one of the vertices of the primitive that produced the fragment.

For a polygon with more than three edges, such as are produced by clipping a triangle, a convex combination of the values of the datum at the polygon’s vertices must be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

\[
f = \sum_{i=1}^{n} a_i f_i
\]
where \( n \) is the number of vertices in the polygon and \( f_i \) is the value of \( f \) at vertex \( i \). For each \( i \), \( 0 \leq a_i \leq 1 \) and \( \sum_{i=1}^{n} a_i = 1 \). The values of \( a_i \) may differ from fragment to fragment, but at vertex \( i \), \( a_i = 1 \) and \( a_j = 0 \) for \( j \neq i \).

Note

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolated data across each horizontal span from edge to edge also satisfies the restrictions (in this case the numerator and denominator of perspective interpolation are iterated independently, and a division is performed for each fragment).

27.12.2. Polygon Mode

Possible values of the \( 	ext{VkPipelineRasterizationStateCreateInfo}::\text{polygonMode} \) property of the currently active pipeline, specifying the method of rasterization for polygons, are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkPolygonMode {
    VK_POLYGON_MODE_FILL = 0,
    VK_POLYGON_MODE_LINE = 1,
    VK_POLYGON_MODE_POINT = 2,
    // Provided by VK_NV_fill_rectangle
    VK_POLYGON_MODE_FILL_RECTANGLE_NV = 1000153000,
} VkPolygonMode;
```

- \( 	ext{VK_POLYGON_MODE_POINT} \) specifies that polygon vertices are drawn as points.
- \( 	ext{VK_POLYGON_MODE_LINE} \) specifies that polygon edges are drawn as line segments.
- \( 	ext{VK_POLYGON_MODE_FILL} \) specifies that polygons are rendered using the polygon rasterization rules in this section.
- \( 	ext{VK_POLYGON_MODE_FILL_RECTANGLE_NV} \) specifies that polygons are rendered using polygon rasterization rules, modified to consider a sample within the primitive if the sample location is inside the axis-aligned bounding box of the triangle after projection. Note that the barycentric weights used in attribute interpolation can extend outside the range \([0,1]\) when these primitives are shaded. Special treatment is given to a sample position on the boundary edge of the bounding box. In such a case, if two rectangles lie on either side of a common edge (with identical endpoints) on which a sample position lies, then exactly one of the triangles must produce a fragment that covers that sample during rasterization.

Polygons rendered in \( 	ext{VK_POLYGON_MODE_FILL_RECTANGLE_NV} \) mode may be clipped by the frustum or by user clip planes. If clipping is applied, the triangle is culled rather than clipped.

Area calculation and facingness are determined for \( 	ext{VK_POLYGON_MODE_FILL_RECTANGLE_NV} \) mode using the triangle’s vertices.
These modes affect only the final rasterization of polygons: in particular, a polygon's vertices are shaded and the polygon is clipped and possibly culled before these modes are applied.

27.12.3. Depth Bias

The depth values of all fragments generated by the rasterization of a polygon can be offset by a single value that is computed for that polygon. This behavior is controlled by the depthBiasEnable, depthBiasConstantFactor, depthBiasClamp, and depthBiasSlopeFactor members of VkPipelineRasterizationStateCreateInfo, or by the corresponding parameters to the vkCmdSetDepthBias and vkCmdSetDepthBiasEnableEXT command if depth bias state is dynamic.

```c
// Provided by VK_VERSION_1_0
void vkCmdSetDepthBias(
    VkCommandBuffer                             commandBuffer,
    float                                       depthBiasConstantFactor,
    float                                       depthBiasClamp,
    float                                       depthBiasSlopeFactor);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **depthBiasConstantFactor** is a scalar factor controlling the constant depth value added to each fragment.
- **depthBiasClamp** is the maximum (or minimum) depth bias of a fragment.
- **depthBiasSlopeFactor** is a scalar factor applied to a fragment's slope in depth bias calculations.

If depthBiasEnable is VK_FALSE at draw time, no depth bias is applied and the fragment's depth values are unchanged.

depthBiasSlopeFactor scales the maximum depth slope of the polygon, and depthBiasConstantFactor scales the minimum resolvable difference of the depth buffer. The resulting values are summed to produce the depth bias value which is then clamped to a minimum or maximum value specified by depthBiasClamp. depthBiasSlopeFactor, depthBiasConstantFactor, and depthBiasClamp can each be positive, negative, or zero.

The maximum depth slope \( m \) of a triangle is

\[
m = \sqrt{\left(\frac{\partial z_f}{\partial x_f}\right)^2 + \left(\frac{\partial z_f}{\partial y_f}\right)^2}
\]

where \((x_0, y_0, z_0)\) is a point on the triangle. \( m \) may be approximated as

\[
m = \max\left(\frac{\partial z_f}{\partial x_f}, \frac{\partial z_f}{\partial y_f}\right).
\]

The minimum resolvable difference \( r \) is a parameter that depends on the depth buffer representation. It is the smallest difference in framebuffer coordinate \( z \) values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but \( z \) values that differ by \( r \), will have distinct depth values.
For fixed-point depth buffer representations, \( r \) is constant throughout the range of the entire depth buffer. Its value is implementation dependent but must be at most

\[
    r = 2 \times 2^n
\]

for an \( n \)-bit buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, \( e \), in the range of \( z \) values spanned by the primitive. If \( n \) is the number of bits in the floating-point mantissa, the minimum resolvable difference, \( r \), for the given primitive is defined as

\[
    r = 2^e^n
\]

If a triangle is rasterized using the VK_POLYGON_MODE_FILL_RECTANGLE_NV polygon mode, then this minimum resolvable difference may not be resolvable for samples outside of the triangle, where the depth is extrapolated.

If no depth buffer is present, \( r \) is undefined.

The bias value \( o \) for a polygon is

\[
    o = \text{dbclamp}(m \times \text{depthBiasSlopeFactor} + r \times \text{depthBiasConstantFactor})
\]

where

\[
    \text{dbclamp}(x) = \begin{cases} 
    x & \text{depthBiasClamp} = 0 \text{ or }NaN \\
    \min(x, \text{depthBiasClamp}) & \text{depthBiasClamp} > 0 \\
    \max(x, \text{depthBiasClamp}) & \text{depthBiasClamp} < 0 
\end{cases}
\]

\( m \) is computed as described above. If the depth buffer uses a fixed-point representation, \( m \) is a function of depth values in the range \([0,1]\), and \( o \) is applied to depth values in the same range.

For fixed-point depth buffers, fragment depth values are always limited to the range \([0,1]\) by clamping after depth bias addition is performed. Unless the VK_EXT_depth_range_unrestricted extension is enabled, fragment depth values are clamped even when the depth buffer uses a floating-point representation.

Valid Usage

- VUID-vkCmdSetDepthBias-depthBiasClamp-00790
  If the depth bias clamping feature is not enabled, depthBiasClamp must be 0.0
Valid Usage (Implicit)

- VUID-vkCmdSetDepthBias-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthBias-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetDepthBias-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically control whether to bias fragment depth values:

```c
// Provided by VK_EXT_extended_dynamic_state2
void vkCmdSetDepthBiasEnableEXT(
    VkCommandBuffer              commandBuffer,    commandBuffer,
    VkBool32                     depthBiasEnable);    depthBiasEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthBiasEnable` controls whether to bias fragment depth values.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

Valid Usage

- VUID-vkCmdSetDepthBiasEnableEXT-None-04872
  
  The `extendedDynamicState2` feature must be enabled
27.12.4. Conservative Rasterization

Polygon rasterization can be made conservative by setting `conservativeRasterizationMode` to `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` or `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` in `VkPipelineRasterizationConservativeStateCreateInfoEXT`. The state is set by adding this structure to the `pNext` chain of a `VkPipelineRasterizationStateCreateInfo` structure when creating the graphics pipeline. Enabling these modes also affects line and point rasterization if the implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization` to `VK_TRUE`.

`VkPipelineRasterizationConservativeStateCreateInfoEXT` is defined as:
// Provided by VK_EXT_conservative_rasterization

typedef struct VkPipelineRasterizationConservativeStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationConservativeStateCreateFlagsEXT flags;
    VkConservativeRasterizationModeEXT conservativeRasterizationMode;
    float extraPrimitiveOverestimationSize;
} VkPipelineRasterizationConservativeStateCreateInfoEXT;

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is reserved for future use.
• `conservativeRasterizationMode` is the conservative rasterization mode to use.
• `extraPrimitiveOverestimationSize` is the extra size in pixels to increase the generating primitive during conservative rasterization at each of its edges in X and Y equally in screen space beyond the base overestimation specified in `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::primitiveOverestimationSize`.

Valid Usage

• `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-extraPrimitiveOverestimationSize-01769` extraPrimitiveOverestimationSize must be in the range of 0.0 to `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::maxExtraPrimitiveOverestimationSize` inclusive

Valid Usage (Implicit)

• `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-sType-sType` sType must be `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT`
• `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-flags-zero bitmask` flags must be 0
• `VUID-VkPipelineRasterizationConservativeStateCreateInfoEXT-conservativeRasterizationMode-parameter` conservativeRasterizationMode must be a valid `VkConservativeRasterizationModeEXT` value

// Provided by VK_EXT_conservative_rasterization
typedef VkFlags VkPipelineRasterizationConservativeStateCreateFlagsEXT;

VkPipelineRasterizationConservativeStateCreateFlagsEXT is a bitmask type for setting a mask, but is
Currently reserved for future use.

Possible values of `VkPipelineRasterizationConservativeStateCreateInfoEXT::conservativeRasterizationMode`, specifying the conservative rasterization mode are:

```c
// Provided by VK_EXT_conservative_rasterization
typedef enum VkConservativeRasterizationModeEXT {
    VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT = 0,
    VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT = 1,
    VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT = 2,
} VkConservativeRasterizationModeEXT;
```

- **VK_CONSERVATIVE_RASTERIZATION_MODE_DISABLED_EXT** specifies that conservative rasterization is disabled and rasterization proceeds as normal.
- **VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT** specifies that conservative rasterization is enabled in overestimation mode.
- **VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT** specifies that conservative rasterization is enabled in underestimation mode.

When overestimate conservative rasterization is enabled, rather than evaluating coverage at individual sample locations, a determination is made of whether any portion of the pixel (including its edges and corners) is covered by the primitive. If any portion of the pixel is covered, then all bits of the **coverage mask** for the fragment corresponding to that pixel are enabled. If the render pass has a fragment density map attachment and any bit of the **coverage mask** for the fragment is enabled, then all bits of the **coverage mask** for the fragment are enabled.

If the implementation supports `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` and the **PostDepthCoverage** execution mode is specified the **SampleMask** built-in input variable will reflect the coverage after the early per-fragment depth and stencil tests are applied.

For the purposes of evaluating which pixels are covered by the primitive, implementations can increase the size of the primitive by up to `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::primitiveOverestimationSize` pixels at each of the primitive edges. This may increase the number of fragments generated by this primitive and represents an overestimation of the pixel coverage.

This overestimation size can be increased further by setting the **extraPrimitiveOverestimationSize** value above 0.0 in steps of `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::extraPrimitiveOverestimationSizeGranularity` up to and including `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::extraPrimitiveOverestimationSize`. This will further increase the number of fragments generated by this primitive.

The actual precision of the overestimation size used for conservative rasterization may vary between implementations and produce results that only approximate the **primitiveOverestimationSize** and **extraPrimitiveOverestimationSizeGranularity** properties. Implementations may especially vary these approximations when the render pass has a fragment density map and the fragment area covers multiple pixels.
For triangles if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, fragments will be generated if the primitive area covers any portion of any pixel inside the fragment area, including their edges or corners. The tie-breaking rule described in Basic Polygon Rasterization does not apply during conservative rasterization and coverage is set for all fragments generated from shared edges of polygons. Degenerate triangles that evaluate to zero area after rasterization, even for pixels containing a vertex or edge of the zero-area polygon, will be culled if `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::degenerateTrianglesRasterized` is `VK_FALSE` or will generate fragments if `degenerateTrianglesRasterized` is `VK_TRUE`. The fragment input values for these degenerate triangles take their attribute and depth values from the provoking vertex. Degenerate triangles are considered backfacing and the application can enable backface culling if desired. Triangles that are zero area before rasterization may be culled regardless.

For lines if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, and the implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization` to `VK_TRUE`, fragments will be generated if the line covers any portion of any pixel inside the fragment area, including their edges or corners. Degenerate lines that evaluate to zero length after rasterization will be culled if `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::degenerateLinesRasterized` is `VK_FALSE` or will generate fragments if `degenerateLinesRasterized` is `VK_TRUE`. The fragments input values for these degenerate lines take their attribute and depth values from the provoking vertex. Lines that are zero length before rasterization may be culled regardless.

For points if `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT` is enabled, and the implementation sets `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativePointAndLineRasterization` to `VK_TRUE`, fragments will be generated if the point square covers any portion of any pixel inside the fragment area, including their edges or corners.

When underestimate conservative rasterization is enabled, rather than evaluating coverage at individual sample locations, a determination is made of whether all of the pixel (including its edges and corners) is covered by the primitive. If the entire pixel is covered, then a fragment is generated with all bits of its coverage mask corresponding to the pixel enabled, otherwise the pixel is not considered covered even if some portion of the pixel is covered. The fragment is discarded if no pixels inside the fragment area are considered covered. If the render pass has a fragment density map attachment and any pixel inside the fragment area is not considered covered, then the fragment is discarded even if some pixels are considered covered.

If the implementation supports `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` and the PostDepthCoverage execution mode is specified the SampleMask built-in input variable will reflect the coverage after the early per-fragment depth and stencil tests are applied.

For triangles, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if any pixel inside the fragment area is fully covered by the generating primitive, including its edges and corners.

For lines, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will be generated if any pixel inside the fragment area, including its edges and corners, are entirely covered by the line.
For points, if `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if the point square covers the entirety of any pixel square inside the fragment area, including its edges or corners.

If the render pass has a fragment density map and `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` is enabled, fragments will only be generated if the entirety of all pixels inside the fragment area are covered by the generating primitive, line, or point.

For both overestimate and underestimate conservative rasterization modes a fragment has all of its pixel squares fully covered by the generating primitive **must** set `FullyCoveredEXT` to `VK_TRUE` if the implementation enables the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::fullyCoveredFragmentShaderInputVariable` feature.

When the use of a shading rate image or setting the fragment shading rate results in fragments covering multiple pixels, coverage for conservative rasterization is still evaluated on a per-pixel basis and may result in fragments with partial coverage. For fragment shader inputs decorated with `FullyCoveredEXT`, a fragment is considered fully covered if and only if all pixels in the fragment are fully covered by the generating primitive.
Chapter 28. Fragment Operations

Fragments produced by rasterization go through a number of operations to determine whether or how values produced by fragment shading are written to the framebuffer.

The following fragment operations adhere to rasterization order, and are typically performed in this order:

1. Discard rectangles test
2. Scissor test
3. Exclusive scissor test
4. Sample mask test
5. Fragment shading
6. Multisample coverage
7. Depth bounds test
8. Stencil test
9. Depth test
10. Representative fragment test
11. Sample counting
12. Coverage to color
13. Coverage reduction
14. Coverage modulation

The coverage mask generated by rasterization describes the initial coverage of each sample covered by the fragment. Fragment operations will update the coverage mask to add or subtract coverage where appropriate. If a fragment operation results in all bits of the coverage mask being 0, the fragment is discarded, and no further operations are performed. Fragments can also be programmatically discarded in a fragment shader by executing one of

- OpTerminateInvocation
- OpDemoteToHelperInvocationEXT
- OpKill.

When one of the fragment operations in this chapter is described as “replacing” a fragment shader output, that output is replaced unconditionally, even if no fragment shader previously wrote to that output.

If post-depth coverage is enabled, the sample mask test is instead performed after the depth test.

If early per-fragment operations are enabled, fragment shading and multisample coverage operations are instead performed after sample counting.

Once all fragment operations have completed, fragment shader outputs for covered color
attachment samples pass through framebuffer operations.

28.1. Discard Rectangles Test

The discard rectangle test compares the framebuffer coordinates \((x_f, y_f)\) of each sample covered by a fragment against a set of discard rectangles.

Each discard rectangle is defined by a \(VkRect2D\). These values are either set by the \(VkPipelineDiscardRectangleStateCreateInfoEXT\) structure during pipeline creation, or dynamically by the \(vkCmdSetDiscardRectangleEXT\) command.

A given sample is considered inside a discard rectangle if the \(x_f\) is in the range \([VkRect2D::offset.x, VkRect2D::offset.x + VkRect2D::extent.x)\), and \(y_f\) is in the range \([VkRect2D::offset.y, VkRect2D::offset.y + VkRect2D::extent.y)\). If the test is set to be inclusive, samples that are not inside any of the discard rectangles will have their coverage set to 0. If the test is set to be exclusive, samples that are inside any of the discard rectangles will have their coverage set to 0.

If no discard rectangles are specified, the coverage mask is unmodified by this operation.

The \(VkPipelineDiscardRectangleStateCreateInfoEXT\) structure is defined as:

```c
// Provided by VK_EXT_discard_rectangles
typedef struct VkPipelineDiscardRectangleStateCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkPipelineDiscardRectangleStateCreateFlagsEXT flags;
    VkDiscardRectangleModeEXT discardRectangleMode;
    uint32_t discardRectangleCount;
    const VkRect2D* pDiscardRectangles;
} VkPipelineDiscardRectangleStateCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **discardRectangleMode** is a \(VkDiscardRectangleModeEXT\) value determining whether the discard rectangle test is inclusive or exclusive.
- **discardRectangleCount** is the number of discard rectangles to use.
- **pDiscardRectangles** is a pointer to an array of \(VkRect2D\) structures defining discard rectangles.

If the \(VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT\) dynamic state is enabled for a pipeline, the **pDiscardRectangles** member is ignored.

When this structure is included in the **pNext** chain of \(VkGraphicsPipelineCreateInfo\), it defines parameters of the discard rectangle test. If this structure is not included in the **pNext** chain, it is equivalent to specifying this structure with a **discardRectangleCount** of 0.
Valid Usage

- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-discardRectangleCount-00582**
  - `discardRectangleCount` must be less than or equal to `VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles`

Valid Usage (Implicit)

- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT`
- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-flags-zerobitmask**
  - `flags` must be 0
- **VUID-VkPipelineDiscardRectangleStateCreateInfoEXT-discardRectangleMode-parameter**
  - `discardRectangleMode` must be a valid `VkDiscardRectangleModeEXT` value

```c
// Provided by VK_EXT_discard_rectangles
typedef VkFlags VkPipelineDiscardRectangleStateCreateFlagsEXT;
```

`VkPipelineDiscardRectangleStateCreateFlagsEXT` is a bitmask type for setting a mask, but is currently reserved for future use.

`VkDiscardRectangleModeEXT` values are:

```c
// Provided by VK_EXT_discard_rectangles
typedef enum VkDiscardRectangleModeEXT {
    VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT = 0,
    VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT = 1,
} VkDiscardRectangleModeEXT;
```

- **VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT** specifies that the discard rectangle test is inclusive.
- **VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT** specifies that the discard rectangle test is exclusive.

The discard rectangles can be set dynamically with the command:

```c
// Provided by VK_EXT_discard_rectangles
void vkCmdSetDiscardRectangleEXT(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstDiscardRectangle,
    uint32_t                                    discardRectangleCount,
    const VkRect2D*                             pDiscardRectangles);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstDiscardRectangle` is the index of the first discard rectangle whose state is updated by the command.
- `discardRectangleCount` is the number of discard rectangles whose state are updated by the command.
- `pDiscardRectangles` is a pointer to an array of `VkRect2D` structures specifying discard rectangles.

The discard rectangle taken from element i of `pDiscardRectangles` replace the current state for the discard rectangle at index `firstDiscardRectangle + i`, for i in [0, `discardRectangleCount`).

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage

- VUID-vkCmdSetDiscardRectangleEXT-firstDiscardRectangle-00585
  The sum of `firstDiscardRectangle` and `discardRectangleCount` must be less than or equal to `VkPhysicalDeviceDiscardRectanglePropertiesEXT::maxDiscardRectangles`

- VUID-vkCmdSetDiscardRectangleEXT-x-00587
  The x and y member of `offset` in each `VkRect2D` element of `pDiscardRectangles` must be greater than or equal to 0.

- VUID-vkCmdSetDiscardRectangleEXT-offset-00588
  Evaluation of `(offset.x + extent.width)` in each `VkRect2D` element of `pDiscardRectangles` must not cause a signed integer addition overflow.

- VUID-vkCmdSetDiscardRectangleEXT-offset-00589
  Evaluation of `(offset.y + extent.height)` in each `VkRect2D` element of `pDiscardRectangles` must not cause a signed integer addition overflow.

- VUID-vkCmdSetDiscardRectangleEXT-viewportScissor2D-04788
  If this command is recorded in a secondary command buffer with `VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D` enabled, then this function must not be called.
Valid Usage (Implicit)

- **VUID-vkCmdSetDiscardRectangleEXT-commandBuffer-parameter**
  
  **commandBuffer** must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdSetDiscardRectangleEXT-pDiscardRectangles-parameter**
  
  **pDiscardRectangles** must be a valid pointer to an array of `discardRectangleCount` `VkRect2D` structures.

- **VUID-vkCmdSetDiscardRectangleEXT-commandBuffer-recording**
  
  **commandBuffer** must be in the recording state.

- **VUID-vkCmdSetDiscardRectangleEXT-commandBuffer-cmdpool**
  
  The `VkCommandPool` that **commandBuffer** was allocated from must support graphics operations.

- **VUID-vkCmdSetDiscardRectangleEXT-discardRectangleCount-arraylength**
  
  **discardRectangleCount** must be greater than 0.

Host Synchronization

- Host access to **commandBuffer** must be externally synchronized.

- Host access to the `VkCommandPool` that **commandBuffer** was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

28.2. Scissor Test

The scissor test compares the framebuffer coordinates \((x_f, y_f)\) of each sample covered by a fragment against a scissor rectangle at the index equal to the fragment’s `ViewportIndex`.

Each scissor rectangle is defined by a `VkRect2D`. These values are either set by the `VkPipelineViewportStateCreateInfo` structure during pipeline creation, or dynamically by the `vkCmdSetScissor` command.

A given sample is considered inside a scissor rectangle if \(x_f\) is in the range \([\text{VkRect2D}::\text{offset}.x, \text{VkRect2D}::\text{offset}.x + \text{VkRect2D}::\text{extent}.x)\), and \(y_f\) is in the range \([\text{VkRect2D}::\text{offset}.y, \text{VkRect2D}::\text{offset}.y + \text{VkRect2D}::\text{extent}.y)\). Samples with coordinates outside the scissor rectangle at the corresponding `ViewportIndex` will have their coverage set to 0.
If a render pass transform is enabled, the (offset.x and offset.y) and (extent.width and extent.height) values are transformed as described in render pass transform before participating in the scissor test.

The scissor rectangles can be set dynamically with the command:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetScissor(
    VkCommandBuffer commandBuffer,
    uint32_t firstScissor,
    uint32_t scissorCount,
    const VkRect2D* pScissors);
```

- **commandBuffer** is the command buffer into which the command will be recorded.
- **firstScissor** is the index of the first scissor whose state is updated by the command.
- **scissorCount** is the number of scissors whose rectangles are updated by the command.
- **pScissors** is a pointer to an array of VkRect2D structures defining scissor rectangles.

The scissor rectangles taken from element i of pScissors replace the current state for the scissor index firstScissor + i, for i in [0, scissorCount).

This command sets the state for a given draw when the graphics pipeline is created with VK_DYNAMIC_STATE_SCISSOR set in VkPipelineDynamicStateCreateInfo::pDynamicStates.
Valid Usage

- VUID-vkCmdSetScissor-firstScissor-00592
  The sum of firstScissor and scissorCount must be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive

- VUID-vkCmdSetScissor-firstScissor-00593
  If the multiple viewports feature is not enabled, firstScissor must be 0

- VUID-vkCmdSetScissor-scissorCount-00594
  If the multiple viewports feature is not enabled, scissorCount must be 1

- VUID-vkCmdSetScissor-x-00595
  The x and y members of offset member of any element of pScissors must be greater than or equal to 0

- VUID-vkCmdSetScissor-offset-00596
  Evaluation of (offset.x + extent.width) must not cause a signed integer addition overflow for any element of pScissors

- VUID-vkCmdSetScissor-offset-00597
  Evaluation of (offset.y + extent.height) must not cause a signed integer addition overflow for any element of pScissors

- VUID-vkCmdSetScissor-viewportScissor2D-04789
  If this command is recorded in a secondary command buffer with VkCommandBufferInheritanceViewportScissorInfoNV::viewportScissor2D enabled, then this function must not be called.

Valid Usage (Implicit)

- VUID-vkCmdSetScissor-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdSetScissor-pScissors-parameter
  pScissors must be a valid pointer to an array of scissorCount VkRect2D structures

- VUID-vkCmdSetScissor-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdSetScissor-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

- VUID-vkCmdSetScissor-scissorCount-arraylength
  scissorCount must be greater than 0
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

28.3. Exclusive Scissor Test

The exclusive scissor test compares the framebuffer coordinates \((x_f, y_f)\) of each sample covered by a fragment against an exclusive scissor rectangle at the index equal to the fragment's `ViewportIndex`.

Each exclusive scissor rectangle is defined by a `VkRect2D`. These values are either set by the `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure during pipeline creation, or dynamically by the `vkCmdSetExclusiveScissorNV` command.

A given sample is considered inside an exclusive scissor rectangle if \(x_f\) is in the range \([VkRect2D::offset.x, VkRect2D::offset.x + VkRect2D::extent.x)\), and \(y_f\) is in the range \([VkRect2D::offset.y, VkRect2D::offset.y + VkRect2D::extent.y)\). Samples with coordinates inside the exclusive scissor rectangle at the corresponding `ViewportIndex` will have their coverage set to 0.

If no exclusive scissor rectangles are specified, the coverage mask is unmodified by this operation.

The `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_scissor_exclusive
typedef struct VkPipelineViewportExclusiveScissorStateCreateInfoNV {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           exclusiveScissorCount;
    const VkRect2D*    pExclusiveScissors;
} VkPipelineViewportExclusiveScissorStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `exclusiveScissorCount` is the number of exclusive scissor rectangles.
- `pExclusiveScissors` is a pointer to an array of `VkRect2D` structures defining exclusive scissor rectangles.
rectangles.

If the `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV` dynamic state is enabled for a pipeline, the `pExclusiveScissors` member is ignored.

When this structure is included in the `pNext` chain of `VkGraphicsPipelineCreateInfo`, it defines parameters of the exclusive scissor test. If this structure is not included in the `pNext` chain, it is equivalent to specifying this structure with a `exclusiveScissorCount` of 0.

### Valid Usage

- **VUID-VkPipelineViewportExclusiveScissorStateCreateInfoNV-exclusiveScissorCount-02027**
  
  If the multiple viewports feature is not enabled, `exclusiveScissorCount` must be 0 or 1

- **VUID-VkPipelineViewportExclusiveScissorStateCreateInfoNV-exclusiveScissorCount-02028**
  
  `exclusiveScissorCount` must be less than or equal to `VkPhysicalDeviceLimits::maxViewports`

- **VUID-VkPipelineViewportExclusiveScissorStateCreateInfoNV-exclusiveScissorCount-02029**
  
  `exclusiveScissorCount` must be 0 or greater than or equal to the `viewportCount` member of `VkPipelineViewportStateCreateInfo`

### Valid Usage (Implicit)

- **VUID-VkPipelineViewportExclusiveScissorStateCreateInfoNV-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_EXCLUSIVE_SCISSOR_STATE_CREATE_INFO_NV`

The exclusive scissor rectangles can be set dynamically with the command:

```c
void vkCmdSetExclusiveScissorNV(
    VkCommandBuffer commandBuffer,  // Provided by VK_NV_scissor_exclusive
    uint32_t firstExclusiveScissor,  // commandBuffer, firstExclusiveScissor, exclusiveScissorCount, pExclusiveScissors
    uint32_t exclusiveScissorCount,  // commandBuffer, firstExclusiveScissor, exclusiveScissorCount, pExclusiveScissors
    const VkRect2D* pExclusiveScissors);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `firstExclusiveScissor` is the index of the first exclusive scissor rectangle whose state is updated by the command.
- `exclusiveScissorCount` is the number of exclusive scissor rectangles updated by the command.
- `pExclusiveScissors` is a pointer to an array of `VkRect2D` structures defining exclusive scissor rectangles.

The scissor rectangles taken from element `i` of `pExclusiveScissors` replace the current state for the scissor index `firstExclusiveScissor + i`, for `i` in `[0, exclusiveScissorCount)`.  

Chapter 28. Fragment Operations | 1559
This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage

- **VUID-vkCmdSetExclusiveScissorNV-None-02031**
  
  The exclusive scissor feature **must** be enabled

- **VUID-vkCmdSetExclusiveScissorNV-firstExclusiveScissor-02034**
  
  The sum of `firstExclusiveScissor` and `exclusiveScissorCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive

- **VUID-vkCmdSetExclusiveScissorNV-firstExclusiveScissor-02035**
  
  If the multiple viewports feature is not enabled, `firstExclusiveScissor` **must** be 0

- **VUID-vkCmdSetExclusiveScissorNV-exclusiveScissorCount-02036**
  
  If the multiple viewports feature is not enabled, `exclusiveScissorCount` **must** be 1

- **VUID-vkCmdSetExclusiveScissorNV-x-02037**
  
  The `x` and `y` members of `offset` in each member of `pExclusiveScissors` **must** be greater than or equal to 0

- **VUID-vkCmdSetExclusiveScissorNV-offset-02038**
  
  Evaluation of `(offset.x + extent.width)` for each member of `pExclusiveScissors` **must** not cause a signed integer addition overflow

- **VUID-vkCmdSetExclusiveScissorNV-offset-02039**
  
  Evaluation of `(offset.y + extent.height)` for each member of `pExclusiveScissors` **must** not cause a signed integer addition overflow

### Valid Usage (Implicit)

- **VUID-vkCmdSetExclusiveScissorNV-commandBuffer-parameter**
  
  `commandBuffer` **must** be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetExclusiveScissorNV-pExclusiveScissors-parameter**
  
  `pExclusiveScissors` **must** be a valid pointer to an array of `exclusiveScissorCount` `VkRect2D` structures

- **VUID-vkCmdSetExclusiveScissorNV-commandBuffer-recording**
  
  `commandBuffer` **must** be in the recording state

- **VUID-vkCmdSetExclusiveScissorNV-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations

- **VUID-vkCmdSetExclusiveScissorNV-exclusiveScissorCount-arraylength**
  
  `exclusiveScissorCount` **must** be greater than 0
Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

## Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 28.4. Sample Mask Test

The sample mask test compares the **coverage mask** for a fragment with the **sample mask** defined by `VkPipelineMultisampleStateCreateInfo::pSampleMask`.

Each bit of the coverage mask is associated with a sample index as described in the **rasterization chapter**. If the bit in `VkPipelineMultisampleStateCreateInfo::pSampleMask` which is associated with that same sample index is set to 0, the coverage mask bit is set to 0.

### 28.5. Multisample Coverage

If a fragment shader is active and its entry point's interface includes a built-in output variable decorated with `SampleMask`, but not `OverrideCoverageNV`, the coverage mask is **ANDed** with the bits of the `SampleMask` built-in to generate a new coverage mask. If the `SampleMask` built-in is also decorated with `OverrideCoverageNV`, the coverage mask is replaced with the mask bits set in the shader. If **sample shading** is enabled, bits written to `SampleMask` corresponding to samples that are not being shaded by the fragment shader invocation are ignored. If no fragment shader is active, or if the active fragment shader does not include `SampleMask` in its interface, the coverage mask is not modified.

Next, the fragment alpha value and coverage mask are modified based on the line coverage factor if the `lineRasterizationMode` member of the `VkPipelineRasterizationStateCreateInfo` structure is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT`, and the `alphaToCoverageEnable` and `alphaToOneEnable` members of the `VkPipelineMultisampleStateCreateInfo` structure.

All alpha values in this section refer only to the alpha component of the fragment shader output that has a `Location` and `Index` decoration of zero (see the **Fragment Output Interface** section). If that shader output has an integer or unsigned integer type, then these operations are skipped.

If the `lineRasterizationMode` member of the `VkPipelineRasterizationStateCreateInfo` structure is `VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT` and the fragment came from a line segment,
then the alpha value is replaced by multiplying it by the coverage factor for the fragment computed during smooth line rasterization.

If \texttt{alphaToCoverageEnable} is enabled, a temporary coverage mask is generated where each bit is determined by the fragment’s alpha value, which is ANDed with the fragment coverage mask.

No specific algorithm is specified for converting the alpha value to a temporary coverage mask. It is intended that the number of 1’s in this value be proportional to the alpha value (clamped to \([0,1]\)), with all 1’s corresponding to a value of 1.0 and all 0’s corresponding to 0.0. The algorithm may be different at different framebuffer coordinates.

Note

Using different algorithms at different framebuffer coordinates may help to avoid artifacts caused by regular coverage sample locations.

Finally, if \texttt{alphaToOneEnable} is enabled, each alpha value is replaced by the maximum representable alpha value for fixed-point color buffers, or by 1.0 for floating-point buffers. Otherwise, the alpha values are not changed.

28.6. Depth and Stencil Operations

Pipeline state controlling the depth bounds tests, stencil test, and depth test is specified through the members of the \texttt{VkPipelineDepthStencilStateCreateInfo} structure.

The \texttt{VkPipelineDepthStencilStateCreateInfo} structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineDepthStencilStateCreateInfo {
    VkStructureType                       sType;
    const void*                           pNext;
    VkPipelineDepthStencilStateCreateFlags flags;
    VkBool32                               depthTestEnable;
    VkBool32                               depthWriteEnable;
    VkCompareOp                            depthCompareOp;
    VkBool32                               depthBoundsTestEnable;
    VkBool32                               stencilTestEnable;
    VkStencilOpState                      front;
    VkStencilOpState                      back;
    float                                  minDepthBounds;
    float                                  maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{flags} is reserved for future use.
- \texttt{depthTestEnable} controls whether depth testing is enabled.
• `depthWriteEnable` controls whether `depth writes` are enabled when `depthTestEnable` is `VK_TRUE`. Depth writes are always disabled when `depthTestEnable` is `VK_FALSE`.
• `depthCompareOp` is the comparison operator used in the `depth test`.
• `depthBoundsTestEnable` controls whether `depth bounds testing` is enabled.
• `stencilTestEnable` controls whether `stencil testing` is enabled.
• `front` and `back` control the parameters of the `stencil test`.
• `minDepthBounds` is the minimum depth bound used in the `depth bounds test`.
• `maxDepthBounds` is the maximum depth bound used in the `depth bounds test`.

### Valid Usage

• **VUID-VkPipelineDepthStencilStateCreateInfo-depthBoundsTestEnable-00598**
  If the `depth bounds testing` feature is not enabled, `depthBoundsTestEnable` must be `VK_FALSE`.
• **VUID-VkPipelineDepthStencilStateCreateInfo-separateStencilMaskRef-04453**
  If the `VK_KHR_portability_subset` extension is enabled, and `VkPipelineRasterizationStateCreateInfo::cullMode` is `VK_CULL_MODE_NONE`, the value of `reference` in each of the `VkStencilOpState` structs in `front` and `back` must be the same.

### Valid Usage (Implicit)

• **VUID-VkPipelineDepthStencilStateCreateInfo-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO`.

• **VUID-VkPipelineDepthStencilStateCreateInfo-pNext-pNext**
  `pNext` must be `NULL`.

• **VUID-VkPipelineDepthStencilStateCreateInfo-flags-zerobitmask**
  `flags` must be `0`.

• **VUID-VkPipelineDepthStencilStateCreateInfo-depthCompareOp-parameter**
  `depthCompareOp` must be a valid `VkCompareOp` value.

• **VUID-VkPipelineDepthStencilStateCreateInfo-front-parameter**
  `front` must be a valid `VkStencilOpState` structure.

• **VUID-VkPipelineDepthStencilStateCreateInfo-back-parameter**
  `back` must be a valid `VkStencilOpState` structure.

// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineDepthStencilStateCreateFlags;

`VkPipelineDepthStencilStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.
28.7. Depth Bounds Test

The depth bounds test compares the depth value \( z_a \) in the depth/stencil attachment at each sample’s framebuffer coordinates \((x_f, y_f)\) and sample index \( i \) against a set of depth bounds.

The depth bounds are determined by two floating point values defining a minimum (\( \text{minDepthBounds} \)) and maximum (\( \text{maxDepthBounds} \)) depth value. These values are either set by the \( \text{VkPipelineDepthStencilStateCreateInfo} \) structure during pipeline creation, or dynamically by \( \text{vkCmdSetDepthBoundsTestEnableEXT} \) and \( \text{vkCmdSetDepthBounds} \).

A given sample is considered within the depth bounds if \( z_a \) is in the range \([\text{minDepthBounds}, \text{maxDepthBounds}]\). Samples with depth attachment values outside of the depth bounds will have their coverage set to 0.

If the depth bounds test is disabled, or if there is no depth attachment, the coverage mask is unmodified by this operation.

To dynamically enable or disable the depth bounds test:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetDepthBoundsTestEnableEXT(
    VkCommandBuffer                             commandBuffer,
    VkBool32                                    depthBoundsTestEnable);
```

- \( \text{commandBuffer} \) is the command buffer into which the command will be recorded.
- \( \text{depthBoundsTestEnable} \) specifies if the depth bounds test is enabled.

This command sets the state for a given draw when the graphics pipeline is created with \( \text{VK_DYNAMIC_STATEDEPTH_BOUNDS_TEST_ENABLE_EXT} \) set in \( \text{VkPipelineDynamicStateCreateInfo} \):\( \text{pDynamicStates} \).

### Valid Usage

- \( \text{VUID-vkCmdSetDepthBoundsTestEnableEXT-None-03349} \)
  
  The \text{extendedDynamicState} feature \textbf{must} be enabled

### Valid Usage (Implicit)

- \( \text{VUID-vkCmdSetDepthBoundsTestEnableEXT-commandBuffer-parameter} \)
  
  \text{commandBuffer} \textbf{must} be a valid \text{VkCommandBuffer} handle

- \( \text{VUID-vkCmdSetDepthBoundsTestEnableEXT-commandBuffer-recording} \)
  
  \text{commandBuffer} \textbf{must} be in the \text{recording state}

- \( \text{VUID-vkCmdSetDepthBoundsTestEnableEXT-commandBuffer-cmdpool} \)
  
  The \text{VkCommandPool} that \text{commandBuffer} was allocated from \textbf{must} support graphics operations
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth bounds range values call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetDepthBounds(
    VkCommandBuffer commandBuffer,
    float minDepthBounds,
    float maxDepthBounds);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `minDepthBounds` is the minimum depth bound.
- `maxDepthBounds` is the maximum depth bound.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_BOUNDS` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

Valid Usage

- VUID-vkCmdSetDepthBounds-minDepthBounds-00600
  Unless the `VK_EXT_depth_range_unrestricted` extension is enabled `minDepthBounds` must be between `0.0` and `1.0`, inclusive

- VUID-vkCmdSetDepthBounds-maxDepthBounds-00601
  Unless the `VK_EXT_depth_range_unrestricted` extension is enabled `maxDepthBounds` must be between `0.0` and `1.0`, inclusive
Valid Usage (Implicit)

- **VUID-vkCmdSetDepthBounds-commandBuffer-parameter**
  - `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetDepthBounds-commandBuffer-recording**
  - `commandBuffer` must be in the recording state

- **VUID-vkCmdSetDepthBounds-commandBuffer-cmdpool**
  - The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

28.8. Stencil Test

The stencil test compares the stencil attachment value $s_a$ in the depth/stencil attachment at each sample’s framebuffer coordinates $(x_f,y_f)$ and sample index $i$ against a stencil reference value.

If the render pass has a fragment density map attachment and the fragment covers multiple pixels, there is an implementation-dependent association of coverage samples to stencil attachment samples within the fragment. However, if all samples in the fragment are covered, and the stencil attachment value is updated as a result of this test, all stencil attachment samples will be updated.

If the stencil test is not enabled, as specified by `vkCmdSetStencilTestEnableEXT` or `VkPipelineDepthStencilStateCreateInfo::stencilTestEnable`, or if there is no stencil attachment, the coverage mask is unmodified by this operation.

The stencil test is controlled by one of two sets of stencil-related state, the front stencil state and the back stencil state. Stencil tests and writes use the back stencil state when processing fragments generated by back-facing polygons, and the front stencil state when processing fragments generated by front-facing polygons or any other primitives.

The comparison performed is based on the `VkCompareOp`, compare mask $s_c$, and stencil reference...
value $s_r$ of the relevant state set. The compare mask and stencil reference value are set by either the `VkPipelineDepthStencilStateCreateInfo` structure during pipeline creation, or by the `vkCmdSetStencilCompareMask` and `vkCmdSetStencilReference` commands respectively. The compare operation is set by `VkStencilOpState::compareOp` during pipeline creation.

The stencil reference and attachment values $s_r$ and $s_a$ are each independently combined with the compare mask $s_c$ using a logical AND operation to create masked reference and attachment values $s'_r$ and $s'_a$. $s'_r$ and $s'_a$ are used as A and B, respectively, in the operation specified by `VkCompareOp`.

If the comparison evaluates to false, the coverage for the sample is set to 0.

A new stencil value $s_g$ is generated according to a stencil operation defined by `VkStencilOp` parameters set by `vkCmdSetStencilOpEXT` or `VkPipelineDepthStencilStateCreateInfo`. If the stencil test fails, `failOp` defines the stencil operation used. If the stencil test passes however, the stencil op used is based on the depth test - if it passes, `VkPipelineDepthStencilStateCreateInfo::passOp` is used, otherwise `VkPipelineDepthStencilStateCreateInfo::depthFailOp` is used.

The stencil attachment value $s_a$ is then updated with the generated stencil value $s_g$ according to the write mask $s_w$ defined by `VkPipelineDepthStencilStateCreateInfo::writeMask` as:

$$s_a = (s_a \land \neg s_w) \lor (s_g \land s_w)$$

To dynamically enable or disable the stencil test, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetStencilTestEnableEXT(
    VkCommandBuffer commandBuffer,
    VkBool32 stencilTestEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `stencilTestEnable` specifies if the stencil test is enabled.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

**Valid Usage**

- `VUID-vkCmdSetStencilTestEnableEXT-None-03350`

  The extendedDynamicState feature must be enabled
Valid Usage (Implicit)

- VUID-vkCmdSetStencilTestEnableEXT-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetStencilTestEnableEXT-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetStencilTestEnableEXT-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the stencil operations, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetStencilOpEXT(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    VkStencilOp failOp,
    VkStencilOp passOp,
    VkStencilOp depthFailOp,
    VkCompareOp compareOp);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `faceMask` is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the stencil operation.
- `failOp` is a `VkStencilOp` value specifying the action performed on samples that fail the stencil test.
- `passOp` is a `VkStencilOp` value specifying the action performed on samples that pass both the depth and stencil tests.
• `depthFailOp` is a `VkStencilOp` value specifying the action performed on samples that pass the stencil test and fail the depth test.

• `compareOp` is a `VkCompareOp` value specifying the comparison operator used in the stencil test.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_OP_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage

- **VUID-vkCmdSetStencilOpEXT-None-03351**  
  The `extendedDynamicState` feature must be enabled

### Valid Usage (Implicit)

- **VUID-vkCmdSetStencilOpEXT-commandBuffer-parameter**  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdSetStencilOpEXT-faceMask-parameter**  
  `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values

- **VUID-vkCmdSetStencilOpEXT-faceMask-requiredbitmask**  
  `faceMask` must not be 0

- **VUID-vkCmdSetStencilOpEXT-failOp-parameter**  
  `failOp` must be a valid `VkStencilOp` value

- **VUID-vkCmdSetStencilOpEXT-passOp-parameter**  
  `passOp` must be a valid `VkStencilOp` value

- **VUID-vkCmdSetStencilOpEXT-depthFailOp-parameter**  
  `depthFailOp` must be a valid `VkStencilOp` value

- **VUID-vkCmdSetStencilOpEXT-compareOp-parameter**  
  `compareOp` must be a valid `VkCompareOp` value

- **VUID-vkCmdSetStencilOpEXT-commandBuffer-recording**  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdSetStencilOpEXT-commandBuffer-cmdpool**  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
The `VkStencilOpState` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkStencilOpState {
    VkStencilOp    failOp;
    VkStencilOp    passOp;
    VkStencilOp    depthFailOp;
    VkCompareOp    compareOp;
    uint32_t       compareMask;
    uint32_t       writeMask;
    uint32_t       reference;
} VkStencilOpState;
```

- `failOp` is a `VkStencilOp` value specifying the action performed on samples that fail the stencil test.
- `passOp` is a `VkStencilOp` value specifying the action performed on samples that pass both the depth and stencil tests.
- `depthFailOp` is a `VkStencilOp` value specifying the action performed on samples that pass the stencil test and fail the depth test.
- `compareOp` is a `VkCompareOp` value specifying the comparison operator used in the stencil test.
- `compareMask` selects the bits of the unsigned integer stencil values participating in the stencil test.
- `writeMask` selects the bits of the unsigned integer stencil values updated by the stencil test in the stencil framebuffer attachment.
- `reference` is an integer reference value that is used in the unsigned stencil comparison.
Valid Usage (Implicit)

• VUID-VkStencilOpState-failOp-parameter
  failOp must be a valid VkStencilOp value

• VUID-VkStencilOpState-passOp-parameter
  passOp must be a valid VkStencilOp value

• VUID-VkStencilOpState-depthFailOp-parameter
  depthFailOp must be a valid VkStencilOp value

• VUID-VkStencilOpState-compareOp-parameter
  compareOp must be a valid VkCompareOp value

To dynamically set the stencil compare mask call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetStencilCompareMask(
    VkCommandBuffer commandBuffer,
    VkStencilFaceFlags faceMask,
    uint32_t compareMask);
```

• commandBuffer is the command buffer into which the command will be recorded.

• faceMask is a bitmask of VkStencilFaceFlagBits specifying the set of stencil state for which to update the compare mask.

• compareMask is the new value to use as the stencil compare mask.

This command sets the state for a given draw when the graphics pipeline is created with VK_DYNAMIC_STATE_STENCILCOMPARE_MASK set in VkPipelineDynamicStateCreateInfo::pDynamicStates.

Valid Usage (Implicit)

• VUID-vkCmdSetStencilCompareMask-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdSetStencilCompareMask-faceMask-parameter
  faceMask must be a valid combination of VkStencilFaceFlagBits values

• VUID-vkCmdSetStencilCompareMask-faceMask-required bitmask
  faceMask must not be 0

• VUID-vkCmdSetStencilCompareMask-commandBuffer-recording
  commandBuffer must be in the recording state

• VUID-vkCmdSetStencilCompareMask-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations
Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`VkStencilFaceFlagBits` values are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FACE_FRONT_AND_BACK = 0x00000003,
    VK_STENCIL_FRONT_AND_BACK = VK_STENCIL_FACE_FRONT_AND_BACK,
} VkStencilFaceFlagBits;
```

- `VK_STENCIL_FACE_FRONT_BIT` specifies that only the front set of stencil state is updated.
- `VK_STENCIL_FACE_BACK_BIT` specifies that only the back set of stencil state is updated.
- `VK_STENCIL_FACE_FRONT_AND_BACK` is the combination of `VK_STENCIL_FACE_FRONT_BIT` and `VK_STENCIL_FACE_BACK_BIT`, and specifies that both sets of stencil state are updated.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkStencilFaceFlags;
```

`VkStencilFaceFlags` is a bitmask type for setting a mask of zero or more `VkStencilFaceFlagBits`.

To dynamically set the stencil write mask call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetStencilWriteMask(
    VkCommandBuffer commandBuffer,                     
    VkStencilFaceFlags faceMask,                      
    uint32_t writeMask);                               
```

- `commandBuffer` is the command buffer into which the command will be recorded.
• **faceMask** is a bitmask of **VkStencilFaceFlagBits** specifying the set of stencil state for which to update the write mask, as described above for **vkCmdSetStencilCompareMask**.

• **writeMask** is the new value to use as the stencil write mask.

This command sets the state for a given draw when the graphics pipeline is created with **VK_DYNAMIC_STATE_STENCIL_WRITE_MASK** set in **VkPipelineDynamicStateCreateInfo::pDynamicStates**.

### Valid Usage (Implicit)

- VUID-vkCmdSetStencilWriteMask-commandBuffer-parameter
  - `commandBuffer` must be a valid **VkCommandBuffer** handle

- VUID-vkCmdSetStencilWriteMask-faceMask-parameter
  - `faceMask` must be a valid combination of **VkStencilFaceFlagBits** values

- VUID-vkCmdSetStencilWriteMask-faceMask-requiredbitmask
  - `faceMask` must not be 0

- VUID-vkCmdSetStencilWriteMask-commandBuffer-recording
  - `commandBuffer` must be in the recording state

- VUID-vkCmdSetStencilWriteMask-commandBuffer-cmdpool
  - The **VkCommandPool** that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the **VkCommandPool** that `commandBuffer` was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the stencil reference value call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetStencilReference(
    VkCommandBuffer          commandBuffer,
    VkStencilFaceFlags      faceMask,
    uint32_t                 reference);
```
• **commandBuffer** is the command buffer into which the command will be recorded.
• **faceMask** is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the reference value, as described above for `vkCmdSetStencilCompareMask`.
• **reference** is the new value to use as the stencil reference value.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_STENCIL_REFERENCE` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage (Implicit)
- **VUID-vkCmdSetStencilReference-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- **VUID-vkCmdSetStencilReference-faceMask-parameter**
  
  `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values
- **VUID-vkCmdSetStencilReference-faceMask-requiredbitmask**
  
  `faceMask` must not be 0
- **VUID-vkCmdSetStencilReference-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state
- **VUID-vkCmdSetStencilReference-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization
- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>

Possible values of `VkStencilOpState::compareOp`, specifying the stencil comparison function, are:
• **VK_COMPARE_OP_NEVER** specifies that the test evaluates to false.
• **VK_COMPARE_OP_LESS** specifies that the test evaluates \( A < B \).
• **VK_COMPARE_OP_EQUAL** specifies that the test evaluates \( A = B \).
• **VK_COMPARE_OP_LESS_OR_EQUAL** specifies that the test evaluates \( A \leq B \).
• **VK_COMPARE_OP_GREATER** specifies that the test evaluates \( A > B \).
• **VK_COMPARE_OP_NOT_EQUAL** specifies that the test evaluates \( A \neq B \).
• **VK_COMPARE_OP_GREATER_OR_EQUAL** specifies that the test evaluates \( A \geq B \).
• **VK_COMPARE_OP_ALWAYS** specifies that the test evaluates to true.

Possible values of the `failOp`, `passOp`, and `depthFailOp` members of `VkStencilOpState`, specifying what happens to the stored stencil value if this or certain subsequent tests fail or pass, are:

• **VK_STENCIL_OP_KEEP** keeps the current value.
• **VK_STENCIL_OP_ZERO** sets the value to 0.
• **VK_STENCIL_OP_REPLACE** sets the value to `reference`.
• **VK_STENCIL_OP_INCREMENT_AND_CLAMP** increments the current value and clamps to the maximum representable unsigned value.
• **VK_STENCIL_OP_DECREMENT_AND_CLAMP** decrements the current value and clamps to 0.
• **VK_STENCIL_OP_INVERT** bitwise-inverts the current value.
• **VK_STENCIL_OP_INCREMENT_AND_WRAP** increments the current value and wraps to 0 when the maximum value would have been exceeded.

• **VK_STENCIL_OP_DECREMENT_AND_WRAP** decrements the current value and wraps to the maximum possible value when the value would go below 0.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer.

### 28.9. Depth Test

The depth test compares the depth value \( z_a \) in the depth/stencil attachment at each sample's framebuffer coordinates \((x_f, y_f)\) and sample index \(i\) against the sample's depth value \( z_f \).

If the render pass has a fragment density map attachment and the fragment covers multiple pixels, there is an implementation-dependent association of rasterization samples to depth attachment samples within the fragment. However, if all samples in the fragment are covered, and the depth attachment value is updated as a result of this test, all depth attachment samples will be updated.

If the depth test is not enabled, as specified by `vkCmdSetDepthTestEnableEXT` or `VkPipelineDepthStencilStateCreateInfo::depthTestEnable`, or if there is no depth attachment, the coverage mask is unmodified by this operation.

The comparison performed is based on the `VkCompareOp`, set by `vkCmdSetDepthCompareOpEXT` or `VkPipelineDepthStencilStateCreateInfo::depthCompareOp` during pipeline creation. \( z_f \) and \( z_a \) are used as A and B, respectively, in the operation specified by the `VkCompareOp`.

If `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is enabled, before the sample’s \( z_f \) is compared to \( z_a \), \( z_f \) is clamped to \([\text{min}(n, f), \text{max}(n, f)]\), where \( n \) and \( f \) are the `minDepth` and `maxDepth` depth range values of the viewport used by this fragment, respectively.

If the comparison evaluates to false, the coverage for the sample is set to 0.

If depth writes are enabled, as specified by `vkCmdSetDepthWriteEnableEXT` or `VkPipelineDepthStencilStateCreateInfo::depthWriteEnable`, and the comparison evaluated to true, the depth attachment value \( z_a \) is set to the sample’s depth value \( z_f \).

If the depth attachment has a fixed-point format and \( z_f \) is outside of the range \([0.0, 1.0]\), it is clamped to that range before writing.

To dynamically enable or disable the depth test, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetDepthTestEnableEXT(
    VkCommandBuffer commandBuffer,         // Provided by VK_EXT_extended_dynamic_state
    VkBool32 depthTestEnable);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthTestEnable` specifies if the depth test is enabled.
This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

**Valid Usage**

- VUID-vkCmdSetDepthTestEnableEXT-None-03352
  The `extendedDynamicState` feature must be enabled

**Valid Usage (Implicit)**

- VUID-vkCmdSetDepthTestEnableEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthTestEnableEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetDepthTestEnableEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically set the depth compare operations, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetDepthCompareOpEXT(
    VkCommandBuffer commandBuffer,
    VkCompareOp depthCompareOp);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `depthCompareOp` specifies the depth comparison operator.
This command sets the state for a given draw when the graphics pipeline is created with
VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT set in VkPipelineDynamicStateCreateInfo::pDynamicStates.

### Valid Usage

- VUID-vkCmdSetDepthCompareOpEXT-None-03353
  The extendedDynamicState feature must be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetDepthCompareOpEXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdSetDepthCompareOpEXT-depthCompareOp-parameter
  depthCompareOp must be a valid VkCompareOp value

- VUID-vkCmdSetDepthCompareOpEXT-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdSetDepthCompareOpEXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

### Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To dynamically enable or disable depth writes, call:

```c
// Provided by VK_EXT_extended_dynamic_state
void vkCmdSetDepthWriteEnableEXT(
    VkCommandBuffer                             commandBuffer,
    VkBool32                                    depthWriteEnable);
```

- commandBuffer is the command buffer into which the command will be recorded.
• **depthWriteEnable** specifies if depth writes are enabled.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage

- VUID-vkCmdSetDepthWriteEnableEXT-None-03354
  The extendedDynamicState feature must be enabled

### Valid Usage (Implicit)

- VUID-vkCmdSetDepthWriteEnableEXT-commandBuffer-parameter
  commandBuffer must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetDepthWriteEnableEXT-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdSetDepthWriteEnableEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 28.10. Representative Fragment Test

The representative fragment test allows implementations to reduce the amount of rasterization and fragment processing work performed for each point, line, or triangle primitive. For any primitive that produces one or more fragments that pass all prior early fragment tests, the implementation may choose one or more “representative” fragments for processing and discard all other fragments. For draw calls rendering multiple points, lines, or triangles arranged in lists, strips, or fans, the representative fragment test is performed independently for each of those primitives. The
set of fragments discarded by the representative fragment test is implementation-dependent. In some cases, the representative fragment test may not discard any fragments for a given primitive.

If the `pNext` chain of `VkGraphicsPipelineCreateInfo` includes a `VkPipelineRepresentativeFragmentTestStateCreateInfoNV` structure, then that structure includes parameters that control the representative fragment test.

The `VkPipelineRepresentativeFragmentTestStateCreateInfoNV` structure is defined as:

```c
typedef struct VkPipelineRepresentativeFragmentTestStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkBool32           representativeFragmentTestEnable;
} VkPipelineRepresentativeFragmentTestStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `representativeFragmentTestEnable` controls whether the representative fragment test is enabled.

If this structure is not present, `representativeFragmentTestEnable` is considered to be `VK_FALSE`, and the representative fragment test is disabled.

If early fragment tests are not enabled in the active fragment shader, the representative fragment shader test has no effect, even if enabled.

### Valid Usage (Implicit)

- `VUID-VkPipelineRepresentativeFragmentTestStateCreateInfoNV-sType-sType` must be `VK_STRUCTURE_TYPE_PIPELINE_REPRESENTATIVE_FRAGMENT_TEST_STATE_CREATE_INFO_NV`.

#### 28.11. Sample Counting

Occlusion queries use query pool entries to track the number of samples that pass all the per-fragment tests. The mechanism of collecting an occlusion query value is described in [Occlusion Queries](#).

The occlusion query sample counter increments by one for each sample with a coverage value of 1 in each fragment that survives all the per-fragment tests, including scissor, exclusive scissor, sample mask, alpha to coverage, stencil, and depth tests.

#### 28.12. Fragment Coverage To Color

The `VkPipelineCoverageToColorStateCreateInfoNV` structure is defined as:
typedef struct VkPipelineCoverageToColorStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCoverageToColorStateCreateFlagsNV flags;
    VkBool32 coverageToColorEnable;
    uint32_t coverageToColorLocation;
} VkPipelineCoverageToColorStateCreateInfoNV;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is reserved for future use.
- coverageToColorEnable controls whether the fragment coverage value replaces a fragment color output.
- coverageToColorLocation controls which fragment shader color output value is replaced.

If the pNext chain of VkPipelineMultisampleStateCreateInfo includes a VkPipelineCoverageToColorStateCreateInfoNV structure, then that structure controls whether the fragment coverage is substituted for a fragment color output and, if so, which output is replaced.

If coverageToColorEnable is VK_TRUE, the coverage mask replaces the first component of the color value corresponding to the fragment shader output location with Location equal to coverageToColorLocation and Index equal to zero. If the color attachment format has fewer bits than the coverage mask, the low bits of the sample coverage mask are taken without any clamping. If the color attachment format has more bits than the coverage mask, the high bits of the sample coverage mask are filled with zeros.

If coverageToColorEnable is VK_FALSE, these operations are skipped. If this structure is not present, it is as if coverageToColorEnable is VK_FALSE.

Valid Usage

- VUID-VkPipelineCoverageToColorStateCreateInfoNV-coverageToColorEnable-01404

If coverageToColorEnable is VK_TRUE, then the render pass subpass indicated by VkGraphicsPipelineCreateInfo::renderPass and VkGraphicsPipelineCreateInfo::subpass must have a color attachment at the location selected by coverageToColorLocation, with a VkFormat of VK_FORMAT_R8_UINT, VK_FORMAT_R8_SINT, VK_FORMAT_R16_UINT, VK_FORMAT_R16_SINT, VK_FORMAT_R32_UINT, VK_FORMAT_R32_SINT, or VK_FORMAT_R32_SINT
**Valid Usage (Implicit)**

- VUID-VkPipelineCoverageToColorStateCreateInfoNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_TO_COLOR_STATE_CREATE_INFO_NV`

- VUID-VkPipelineCoverageToColorStateCreateInfoNV-flags-zerobitmask
  
  `flags` must be 0

```c
// Provided by VK_NV_fragment_coverage_to_color
typedef VkFlags VkPipelineCoverageToColorStateCreateFlagsNV;
```

`VkPipelineCoverageToColorStateCreateFlagsNV` is a bitmask type for setting a mask, but is currently reserved for future use.

## 28.13. Coverage Reduction

Coverage reduction takes the coverage information for a fragment and converts that to a boolean coverage value for each color sample in each pixel covered by the fragment.

### 28.13.1. Pixel Coverage

Coverage for each pixel is first extracted from the total fragment coverage mask. This consists of `rasterizationSamples` unique coverage samples for each pixel in the fragment area, each with a unique `sample index`. If the fragment only contains a single pixel, coverage for the pixel is equivalent to the fragment coverage.

If the render pass has a fragment density map attachment and the fragment covers multiple pixels, pixel coverage is generated in an implementation-dependent manner. If all samples in the fragment are covered, all samples will be covered in each pixel coverage.

If a shading rate image is used, and the fragment covers multiple pixels, each pixel's coverage consists of the coverage samples corresponding to that pixel, and each sample retains its unique `sample index i`.

If the fragment shading rate is set, and the fragment covers multiple pixels, each pixel's coverage consists of the coverage samples with a `pixel index` matching that pixel, and each sample retains its unique `sample index i`.

### 28.13.2. Color Sample Coverage

Once pixel coverage is determined, coverage for each individual color sample corresponding to that pixel is determined.

If the number of `rasterizationSamples` is identical to the number of samples in the color attachments. A color sample is covered if the pixel coverage sample with the same `sample index i` is covered.
Otherwise, the coverage for each color sample is computed from the pixel coverage as follows.

If the `VK_AMD_mixed_attachment_samples` extension is enabled, for color samples present in the color attachments, a color sample is covered if the pixel coverage sample with the same sample index i is covered; additional pixel coverage samples are discarded.

When the `VK_NV_coverage_reduction_mode` extension is enabled, the pipeline state controlling coverage reduction is specified through the members of the `VkPipelineCoverageReductionStateCreateInfoNV` structure.

The `VkPipelineCoverageReductionStateCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_coverage_reduction_mode
typedef struct VkPipelineCoverageReductionStateCreateInfoNV {
    VkStructureType                                  sType;
    const void*                                      pNext;
    VkPipelineCoverageReductionStateCreateFlagsNV    flags;
    VkCoverageReductionModeNV                        coverageReductionMode;
} VkPipelineCoverageReductionStateCreateInfoNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `coverageReductionMode` is a `VkCoverageReductionModeNV` value controlling how color sample coverage is generated from pixel coverage.

If this structure is not present, or if the extension is not enabled, the default coverage reduction mode is inferred as follows:

- If the `VK_NV_framebuffer_mixed_samples` extension is enabled, then it is as if the `coverageReductionMode` is `VK_COVERAGE_REDUCTION_MODE_MERGE_NV`.
- If the `VK_AMD_mixed_attachment_samples` extension is enabled, then it is as if the `coverageReductionMode` is `VK_COVERAGE_REDUCTION_MODE_TRUNCATE_NV`.
- If both `VK_NV_framebuffer_mixed_samples` and `VK_AMD_mixed_attachment_samples` are enabled, then the default coverage reduction mode is implementation-dependent.

**Valid Usage (Implicit)**

- `VUID-VkPipelineCoverageReductionStateCreateInfoNV-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_REDUCTION_STATE_CREATE_INFO_NV`
- `VUID-VkPipelineCoverageReductionStateCreateInfoNV-flags-zerobitmask`  
  `flags` must be `0`
- `VUID-VkPipelineCoverageReductionStateCreateInfoNV-coverageReductionMode-parameter`  
  `coverageReductionMode` must be a valid `VkCoverageReductionModeNV` value

Chapter 28. Fragment Operations | 1583
VkPipelineCoverageReductionStateCreateFlagsNV is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of VkPipelineCoverageReductionStateCreateInfoNV::coverageReductionMode, specifying how color sample coverage is generated from pixel coverage, are:

- VK_COVERAGE_REDUCTION_MODE_MERGE_NV specifies that each color sample will be associated with an implementation-dependent subset of samples in the pixel coverage. If any of those associated samples are covered, the color sample is covered.

- VK_COVERAGE_REDUCTION_MODE_TRUNCATE_NV specifies that for color samples present in the color attachments, a color sample is covered if the pixel coverage sample with the same sample index \( i \) is covered; other pixel coverage samples are discarded.

To query the set of mixed sample combinations of coverage reduction mode, rasterization samples and color, depth, stencil attachment sample counts that are supported by a physical device, call:

```
VkResult vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV(
    VkPhysicalDevice physicalDevice,
    uint32_t* pCombinationCount,
    VkFramebufferMixedSamplesCombinationNV* pCombinations);
```

- `physicalDevice` is the physical device from which to query the set of combinations.

- `pCombinationCount` is a pointer to an integer related to the number of combinations available or queried, as described below.

- `pCombinations` is either `NULL` or a pointer to an array of VkFramebufferMixedSamplesCombinationNV values, indicating the supported combinations of coverage reduction mode, rasterization samples, and color, depth, stencil attachment sample counts.

If `pCombinations` is `NULL`, then the number of supported combinations for the given `physicalDevice` is returned in `pCombinationCount`. Otherwise, `pCombinationCount` must point to a variable set by the user to the number of elements in the `pCombinations` array, and on return the variable is overwritten with the number of values actually written to `pCombinations`. If the value of `pCombinationCount` is less than the number of combinations supported for the given `physicalDevice`, at most `pCombinationCount` values will be written to `pCombinations`, and `VK_INCOMPLETE` will be
returned instead of \textbf{VK\_SUCCESS}, to indicate that not all the supported values were returned.

\begin{itemize}
\item Valid Usage (Implicit)
\begin{itemize}
\item VUID-vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV-physicalDevice-parameter
\textit{physicalDevice} \textbf{must} be a valid \texttt{VkPhysicalDevice} handle
\item VUID-vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV-pCombinationCount-parameter
\textit{pCombinationCount} \textbf{must} be a valid pointer to a \texttt{uint32\_t} value
\item VUID-vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV-pCombinations-parameter
\textbf{If} the value referenced by \textit{pCombinationCount} \textbf{is not} 0, and \textit{pCombinations} is not \texttt{NULL}, \textit{pCombinations} \textbf{must} be a valid pointer to an array of \textit{pCombinationCount} \texttt{VkFramebufferMixedSamplesCombinationNV} structures
\end{itemize}
\end{itemize}

\begin{itemize}
\item Return Codes
\begin{itemize}
\item Success
\begin{itemize}
\item \textbf{VK\_SUCCESS}
\item \textbf{VK\_INCOMPLETE}
\end{itemize}
\item Failure
\begin{itemize}
\item \textbf{VK\_ERROR\_OUT\_OF\_HOST\_MEMORY}
\item \textbf{VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY}
\end{itemize}
\end{itemize}
\end{itemize}

\textbf{The} \texttt{VkFramebufferMixedSamplesCombinationNV} \textbf{structure is defined as:}

\begin{verbatim}
// Provided by VK_NV_coverage_reduction_mode
typedef struct VkFramebufferMixedSamplesCombinationNV {
    VkStructureType sType;
    void* pNext;
    VkCoverageReductionModeNV coverageReductionMode;
    VkSampleCountFlagBits rasterizationSamples;
    VkSampleCountFlags depthStencilSamples;
    VkSampleCountFlags colorSamples;
} VkFramebufferMixedSamplesCombinationNV;
\end{verbatim}

\begin{itemize}
\item \textit{sType} is the type of this structure.
\item \textit{pNext} \textbf{is} \texttt{NULL} or a pointer to a structure extending this structure.
\item \textit{coverageReductionMode} \textbf{is} a \texttt{VkCoverageReductionModeNV} \textbf{value specifying} the coverage reduction mode.
\item \textit{rasterizationSamples} \textbf{specifies} the number of rasterization samples in the supported combination.
\item \textit{depthStencilSamples} \textbf{specifies} the number of samples in the depth stencil attachment in the
supported combination. A value of 0 indicates the combination does not have a depth stencil attachment.

- **colorSamples** specifies the number of color samples in a color attachment in the supported combination. A value of 0 indicates the combination does not have a color attachment.

### Valid Usage (Implicit)

- VUID-VkFramebufferMixedSamplesCombinationNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_FRAMEBUFFER_MIXED_SAMPLES_COMBINATION_NV`

- VUID-VkFramebufferMixedSamplesCombinationNV-pNext-pNext
  
  `pNext` must be `NULL`

## 28.13.3. Coverage Modulation

As part of coverage reduction, fragment color values can also be modulated (multiplied) by a value that is a function of fraction of covered rasterization samples associated with that color sample.

Pipeline state controlling coverage modulation is specified through the members of the `VkPipelineCoverageModulationStateCreateInfoNV` structure.

The `VkPipelineCoverageModulationStateCreateInfoNV` structure is defined as:

```c
// Provided by VK_NV_framebuffer_mixed_samples
typedef struct VkPipelineCoverageModulationStateCreateInfoNV {
    VkStructureType sType;
    const void* pNext;
    VkPipelineCoverageModulationStateCreateFlagsNV flags;
    VkCoverageModulationModeNV coverageModulationMode;
    VkBool32 coverageModulationTableEnable;
    uint32_t coverageModulationTableCount;
    const float* pCoverageModulationTable;
} VkPipelineCoverageModulationStateCreateInfoNV;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **coverageModulationMode** is a `VkCoverageModulationModeNV` value controlling which color components are modulated.
- **coverageModulationTableEnable** controls whether the modulation factor is looked up from a table in `pCoverageModulationTable`.
- **coverageModulationTableCount** is the number of elements in `pCoverageModulationTable`.
- **pCoverageModulationTable** is a table of modulation factors containing a value for each number of covered samples.
If `coverageModulationTableEnable` is `VK_FALSE`, then for each color sample the associated bits of the pixel coverage are counted and divided by the number of associated bits to produce a modulation factor \( R \) in the range \((0,1]\) (a value of zero would have been killed due to a color coverage of 0). Specifically:

- \( N = \text{value of } \text{rasterizationSamples} \)
- \( M = \text{value of } \text{VkAttachmentDescription::samples} \) for any color attachments
- \( R = \text{popcount(associated coverage bits)} / (N / M) \)

If `coverageModulationTableEnable` is `VK_TRUE`, the value \( R \) is computed using a programmable lookup table. The lookup table has \( N / M \) elements, and the element of the table is selected by:

- \( R = \text{pCoverageModulationTable}[\text{popcount(associated coverage bits)}-1] \)

Note that the table does not have an entry for \( \text{popcount(associated coverage bits)} = 0 \), because such samples would have been killed.

The values of `pCoverageModulationTable` may be rounded to an implementation-dependent precision, which is at least as fine as \( 1 / N \), and clamped to \([0,1]\).

For each color attachment with a floating point or normalized color format, each fragment output color value is replicated to \( M \) values which can each be modulated (multiplied) by that color sample's associated value of \( R \). Which components are modulated is controlled by `coverageModulationMode`.

If this structure is not present, it is as if `coverageModulationMode` is `VK_COVERAGE_MODULATION_MODE_NONE_NV`.

If the coverage reduction mode is `VK_COVERAGE_REDUCTION_MODE_TRUNCATE_NV`, each color sample is associated with only a single coverage sample. In this case, it is as if `coverageModulationMode` is `VK_COVERAGE_MODULATION_MODE_NONE_NV`.

**Valid Usage**

- VUID-VkPipelineCoverageModulationStateCreateInfoNV-coverageModulationTableEnable-01405
  If `coverageModulationTableEnable` is `VK_TRUE`, `coverageModulationTableCount` must be equal to the number of rasterization samples divided by the number of color samples in the subpass

Chapter 28. Fragment Operations | 1587
Valid Usage (Implicit)

- VUID-VkPipelineCoverageModulationStateCreateInfoNV-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_MODULATION_STATE_CREATE_INFO_NV`

- VUID-VkPipelineCoverageModulationStateCreateInfoNV-flags-zerobitmask
  
  `flags` **must** be 0

- VUID-VkPipelineCoverageModulationStateCreateInfoNV-coverageModulationMode-parameter
  
  `coverageModulationMode` **must** be a valid `VkCoverageModulationModeNV` value

// Provided by VK_NV_framebuffer_mixed_samples

typedef VkFlags VkPipelineCoverageModulationStateCreateFlagsNV;

`VkPipelineCoverageModulationStateCreateFlagsNV` is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of `VkPipelineCoverageModulationStateCreateInfoNV::coverageModulationMode`, specifying which color components are modulated, are:

// Provided by VK_NV_framebuffer_mixed_samples

typedef enum VkCoverageModulationModeNV {
    VK_COVERAGE_MODULATION_MODE_NONE_NV = 0,
    VK_COVERAGE_MODULATION_MODE_RGB_NV = 1,
    VK_COVERAGE_MODULATION_MODE_ALPHA_NV = 2,
    VK_COVERAGE_MODULATION_MODE_RGBA_NV = 3,
} VkCoverageModulationModeNV;

- **VK_COVERAGE_MODULATION_MODE_NONE_NV** specifies that no components are multiplied by the modulation factor.
- **VK_COVERAGE_MODULATION_MODE_RGB_NV** specifies that the red, green, and blue components are multiplied by the modulation factor.
- **VK_COVERAGE_MODULATION_MODE_ALPHA_NV** specifies that the alpha component is multiplied by the modulation factor.
- **VK_COVERAGE_MODULATION_MODE_RGBA_NV** specifies that all components are multiplied by the modulation factor.
Chapter 29. The Framebuffer

29.1. Blending

Blending combines the incoming source fragment’s R, G, B, and A values with the destination R, G, B, and A values of each sample stored in the framebuffer at the fragment’s \((x_f, y_f)\) location. Blending is performed for each color sample covered by the fragment, rather than just once for each fragment.

Source and destination values are combined according to the blend operation, quadruplets of source and destination weighting factors determined by the blend factors, and a blend constant, to obtain a new set of R, G, B, and A values, as described below.

Blending is computed and applied separately to each color attachment used by the subpass, with separate controls for each attachment.

Prior to performing the blend operation, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point as specified by Conversion from Normalized Fixed-Point to Floating-Point. Blending computations are treated as if carried out in floating-point, and basic blend operations are performed with a precision and dynamic range no lower than that used to represent destination components. Advanced blending operations are performed with a precision and dynamic range no lower than the smaller of that used to represent destination components or that used to represent 16-bit floating-point values.

Note

Blending is only defined for floating-point, UNORM, SNORM, and sRGB formats. Within those formats, the implementation may only support blending on some subset of them. Which formats support blending is indicated by VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT.

The pipeline blend state is included in the VkPipelineColorBlendStateCreateInfo structure during graphics pipeline creation:

The VkPipelineColorBlendStateCreateInfo structure is defined as:

```c
typedef struct VkPipelineColorBlendStateCreateInfo {
    VkStructureType                               sType;  // Type of this structure.
    const void*                                   pNext;
    VkPipelineColorBlendStateCreateFlags          flags;  // Flags for this structure.
    VkBool32                                      logicOpEnable;
    VkLogicOp                                     logicOp;
    uint32_t                                      attachmentCount;
    const VkPipelineColorBlendAttachmentState*    pAttachments;
    float                                         blendConstants[4];  // Components for blending.
} VkPipelineColorBlendStateCreateInfo;
```

- **sType** is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is reserved for future use.
• logicOpEnable controls whether to apply Logical Operations.
• logicOp selects which logical operation to apply.
• attachmentCount is the number of VkPipelineColorBlendAttachmentState elements in pAttachments.
• pAttachments is a pointer to an array of per target attachment states.
• blendConstants is a pointer to an array of four values used as the R, G, B, and A components of the blend constant that are used in blending, depending on the blend factor.

Each element of the pAttachments array is a VkPipelineColorBlendAttachmentState structure specifying per-target blending state for each individual color attachment. If the independent blending feature is not enabled on the device, all VkPipelineColorBlendAttachmentState elements in the pAttachments array must be identical.

The value of attachmentCount must be greater than the index of all color attachments that are not VK_ATTACHMENT_UNUSED in VkSubpassDescription::pColorAttachments or VkSubpassDescription2::pColorAttachments for the subpass in which this pipeline is used.

Valid Usage

• VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-00605
  If the independent blending feature is not enabled, all elements of pAttachments must be identical

• VUID-VkPipelineColorBlendStateCreateInfo-logicOpEnable-00606
  If the logic operations feature is not enabled, logicOpEnable must be VK_FALSE

• VUID-VkPipelineColorBlendStateCreateInfo-logicOpEnable-00607
  If logicOpEnable is VK_TRUE, logicOp must be a valid VkLogicOp value
Valid Usage (Implicit)

- **VUID-VkPipelineColorBlendStateCreateInfo-sType-sType**
  
  \[ \textit{sType} \] must be \texttt{VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO}.

- **VUID-VkPipelineColorBlendStateCreateInfo-pNext-pNext**
  
  Each \texttt{pNext} member of any structure (including this one) in the \texttt{pNext} chain must be either \texttt{NULL} or a pointer to a valid instance of \texttt{VkPipelineColorBlendAdvancedStateCreateInfoEXT} or \texttt{VkPipelineColorWriteCreateInfoEXT}.

- **VUID-VkPipelineColorBlendStateCreateInfo-sType-unique**
  
  The \textit{sType} value of each struct in the \texttt{pNext} chain must be unique.

- **VUID-VkPipelineColorBlendStateCreateInfo-flags-zerobitmask**
  
  \textit{flags} must be \texttt{0}.

- **VUID-VkPipelineColorBlendStateCreateInfo-pAttachments-parameter**
  
  If \textit{attachmentCount} is not \texttt{0}, \textit{pAttachments} must be a valid pointer to an array of \textit{attachmentCount} valid \texttt{VkPipelineColorBlendAttachmentState} structures.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkPipelineColorBlendStateCreateFlags;
```

\texttt{VkPipelineColorBlendStateCreateFlags} is a bitmask type for setting a mask, but is currently reserved for future use.

The \texttt{VkPipelineColorBlendAttachmentState} structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPipelineColorBlendAttachmentState {
    VkBool32 blendEnable;
    VkBlendFactor srcColorBlendFactor;
    VkBlendFactor dstColorBlendFactor;
    VkBlendOp colorBlendOp;
    VkBlendFactor srcAlphaBlendFactor;
    VkBlendFactor dstAlphaBlendFactor;
    VkBlendOp alphaBlendOp;
    VkColorComponentFlags colorWriteMask;
} VkPipelineColorBlendAttachmentState;
```

- **blendEnable** controls whether blending is enabled for the corresponding color attachment. If blending is not enabled, the source fragment's color for that attachment is passed through unmodified.

- **srcColorBlendFactor** selects which blend factor is used to determine the source factors (\(S_r,S_g,S_b\)).

- **dstColorBlendFactor** selects which blend factor is used to determine the destination factors (\(D_r,D_g,D_b\)).

- **colorBlendOp** selects which blend operation is used to calculate the RGB values to write to the
• `srcAlphaBlendFactor` selects which blend factor is used to determine the source factor $S_a$.
• `dstAlphaBlendFactor` selects which blend factor is used to determine the destination factor $D_a$.
• `alphaBlendOp` selects which blend operation is used to calculate the alpha values to write to the color attachment.
• `colorWriteMask` is a bitmask of `VkColorComponentFlagBits` specifying which of the R, G, B, and/or A components are enabled for writing, as described for the Color Write Mask.
Valid Usage

- VUID-VkPipelineColorBlendAttachmentState-srcColorBlendFactor-00608
  If the dual source blending feature is not enabled, `srcColorBlendFactor` must not be
  `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`

- VUID-VkPipelineColorBlendAttachmentState-dstColorBlendFactor-00609
  If the dual source blending feature is not enabled, `dstColorBlendFactor` must not be
  `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`

- VUID-VkPipelineColorBlendAttachmentState-srcAlphaBlendFactor-00610
  If the dual source blending feature is not enabled, `srcAlphaBlendFactor` must not be
  `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`

- VUID-VkPipelineColorBlendAttachmentState-dstAlphaBlendFactor-00611
  If the dual source blending feature is not enabled, `dstAlphaBlendFactor` must not be
  `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, or `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA`

- VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-01406
  If either of `colorBlendOp` or `alphaBlendOp` is an advanced blend operation, then `colorBlendOp` must equal `alphaBlendOp`

- VUID-VkPipelineColorBlendAttachmentState-advancedBlendIndependentBlend-01407
  If `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendIndependentBlend` is `VK_FALSE` and `colorBlendOp` is an advanced blend operation, then `colorBlendOp` must be the same for all attachments

- VUID-VkPipelineColorBlendAttachmentState-advancedBlendIndependentBlend-01408
  If `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendIndependentBlend` is `VK_FALSE` and `alphaBlendOp` is an advanced blend operation, then `alphaBlendOp` must be the same for all attachments

- VUID-VkPipelineColorBlendAttachmentState-advancedBlendAllOperations-01409
  If `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendAllOperations` is `VK_FALSE`, then `colorBlendOp` must not be

- VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-01410
  If `colorBlendOp` or `alphaBlendOp` is an advanced blend operation, then `colorAttachmentCount`
of the subpass this pipeline is compiled against must be less than or equal to
\textbf{VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendMaxColorAttachments}

- VUID-VkPipelineColorBlendAttachmentState-constantAlphaColorBlendFactors-04454
  If the \textit{VK_KHR_portability_subset} extension is enabled, and
  \textbf{VkPhysicalDevicePortabilitySubsetFeaturesKHR::constantAlphaColorBlendFactors} is
  \texttt{VK_FALSE}, \texttt{srcColorBlendFactor} must not be \texttt{VK_BLEND_FACTOR_CONSTANT_ALPHA} or
  \texttt{VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA}.

- VUID-VkPipelineColorBlendAttachmentState-constantAlphaColorBlendFactors-04455
  If the \textit{VK_KHR_portability_subset} extension is enabled, and
  \textbf{VkPhysicalDevicePortabilitySubsetFeaturesKHR::constantAlphaColorBlendFactors} is
  \texttt{VK_FALSE}, \texttt{dstColorBlendFactor} must not be \texttt{VK_BLEND_FACTOR_CONSTANT_ALPHA} or
  \texttt{VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA}.

\textbf{Valid Usage (Implicit)}

- VUID-VkPipelineColorBlendAttachmentState-srcColorBlendFactor-parameter
  \texttt{srcColorBlendFactor} must be a valid \texttt{VkBlendFactor} value

- VUID-VkPipelineColorBlendAttachmentState-dstColorBlendFactor-parameter
  \texttt{dstColorBlendFactor} must be a valid \texttt{VkBlendFactor} value

- VUID-VkPipelineColorBlendAttachmentState-colorBlendOp-parameter
  \texttt{colorBlendOp} must be a valid \texttt{VkBlendOp} value

- VUID-VkPipelineColorBlendAttachmentState-srcAlphaBlendFactor-parameter
  \texttt{srcAlphaBlendFactor} must be a valid \texttt{VkBlendFactor} value

- VUID-VkPipelineColorBlendAttachmentState-dstAlphaBlendFactor-parameter
  \texttt{dstAlphaBlendFactor} must be a valid \texttt{VkBlendFactor} value

- VUID-VkPipelineColorBlendAttachmentState-alphaBlendOp-parameter
  \texttt{alphaBlendOp} must be a valid \texttt{VkBlendOp} value

- VUID-VkPipelineColorBlendAttachmentState-colorWriteMask-parameter
  \texttt{colorWriteMask} must be a valid combination of \texttt{VkColorComponentFlagBits} values

\textbf{29.1.1. Blend Factors}

The source and destination color and alpha blending factors are selected from the enum:
typedef enum VkBlendFactor {
    VK_BLEND_FACTOR_ZERO = 0,
    VK_BLEND_FACTOR_ONE = 1,
    VK_BLEND_FACTOR_SRC_COLOR = 2,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR = 3,
    VK_BLEND_FACTOR_DST_COLOR = 4,
    VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR = 5,
    VK_BLEND_FACTOR_SRC_ALPHA = 6,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA = 7,
    VK_BLEND_FACTOR_DST_ALPHA = 8,
    VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA = 9,
    VK_BLEND_FACTOR_CONSTANT_COLOR = 10,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR = 11,
    VK_BLEND_FACTOR_CONSTANT_ALPHA = 12,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA = 13,
    VK_BLEND_FACTOR_SRC_ALPHA_SATURATE = 14,
    VK_BLEND_FACTOR_SRC1_COLOR = 15,
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR = 16,
    VK_BLEND_FACTOR_SRC1_ALPHA = 17,
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA = 18,
} VkBlendFactor;

The semantics of the enum values are described in the table below:

<table>
<thead>
<tr>
<th>VkBlendFactor</th>
<th>RGB Blend Factors (S_r,S_g,S_b) or (D_r,D_g,D_b)</th>
<th>Alpha Blend Factor (S_a or D_a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_FACTOR_ZERO</td>
<td>(0,0,0)</td>
<td>0</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE</td>
<td>(1,1,1)</td>
<td>1</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_COLOR</td>
<td>(R_s,0,G_s,0,B_s)</td>
<td>A_s</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR</td>
<td>(1-R_s,1-G_s,1-B_s)</td>
<td>1-A_s</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_COLOR</td>
<td>(R_d,0,G_d,0,B_d)</td>
<td>A_d</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR</td>
<td>(1-R_d,1-G_d,1-B_d)</td>
<td>1-A_d</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA</td>
<td>(A_s,0,A_s,0,A_s)</td>
<td>A_s</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA</td>
<td>(1-A_s,1-A_s,1-A_s)</td>
<td>1-A_s</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_ALPHA</td>
<td>(A_d,0,A_d,0,A_d)</td>
<td>A_d</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA</td>
<td>(1-A_d,1-A_d,1-A_d)</td>
<td>1-A_d</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_CONSTANT_COLOR</td>
<td>(R_c,G_c,B_c)</td>
<td>A_c</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR</td>
<td>(1-R_c,1-G_c,1-B_c)</td>
<td>1-A_c</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_CONSTANT_ALPHA</td>
<td>(A_c,0,A_c,0,A_c)</td>
<td>A_c</td>
</tr>
</tbody>
</table>
In this table, the following conventions are used:

- $R_{s0}, G_{s0}, B_{s0}$ and $A_{s0}$ represent the first source color $R$, $G$, $B$, and $A$ components, respectively, for the fragment output location corresponding to the color attachment being blended.

- $R_{s1}, G_{s1}, B_{s1}$ and $A_{s1}$ represent the second source color $R$, $G$, $B$, and $A$ components, respectively, used in dual source blending modes, for the fragment output location corresponding to the color attachment being blended.

- $R_d, G_d, B_d$ and $A_d$ represent the $R$, $G$, $B$, and $A$ components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.

- $R_c, G_c, B_c$ and $A_c$ represent the blend constant $R$, $G$, $B$, and $A$ components, respectively.

If the pipeline state object is created without the `VK_DYNAMIC_STATE_BLEND_CONSTANTS` dynamic state enabled then the blend constant $(R_c, G_c, B_c, A_c)$ is specified via the `blendConstants` member of `VkPipelineColorBlendStateCreateInfo`.

Otherwise, to dynamically set and change the blend constant, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdSetBlendConstants(
    VkCommandBuffer commandBuffer,
    const float blendConstants[4]);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `blendConstants` is a pointer to an array of four values specifying the $R$, $G$, $B$, and $A$ components of the blend constant color used in blending, depending on the blend factor.
Valid Usage (Implicit)

- VUID-vkCmdSetBlendConstants-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdSetBlendConstants-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdSetBlendConstants-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics operations

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

29.1.2. Dual-Source Blending

Blend factors that use the secondary color input \((R_{s1}, G_{s1}, B_{s1}, A_{s1})\) (VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR, VK_BLEND_FACTOR_SRC1_ALPHA, and VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA) may consume implementation resources that could otherwise be used for rendering to multiple color attachments. Therefore, the number of color attachments that can be used in a framebuffer may be lower when using dual-source blending.

Dual-source blending is only supported if the dualSrcBlend feature is enabled.

The maximum number of color attachments that can be used in a subpass when using dual-source blending functions is implementation-dependent and is reported as the maxFragmentDualSrcAttachments member of VkPhysicalDeviceLimits.

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of the blender using the Index decoration, as described in Fragment Output Interface. If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.
29.1.3. Blend Operations

Once the source and destination blend factors have been selected, they along with the source and destination components are passed to the blending operations. RGB and alpha components can use different operations. Possible values of `VkBlendOp`, specifying the operations, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkBlendOp {
    VK_BLEND_OP_ADD = 0,
    VK_BLEND_OP_SUBTRACT = 1,
    VK_BLEND_OP_REVERSE_SUBTRACT = 2,
    VK_BLEND_OP_MIN = 3,
    VK_BLEND_OP_MAX = 4,

    // Provided by VK_EXT_blend_operation_advanced
    VK_BLEND_OP_ZERO_EXT = 1000148000,
    VK_BLEND_OP_SRC_EXT = 1000148001,
    VK_BLEND_OP_DST_EXT = 1000148002,
    VK_BLEND_OP_SRC_OVER_EXT = 1000148003,
    VK_BLEND_OP_DST_OVER_EXT = 1000148004,
    VK_BLEND_OP_SRC_IN_EXT = 1000148005,
    VK_BLEND_OP_DST_IN_EXT = 1000148006,
    VK_BLEND_OP_SRC_ATOP_EXT = 1000148009,
    VK_BLEND_OP_DST_ATOP_EXT = 1000148010,
    VK_BLEND_OP_XOR_EXT = 1000148011,
    VK_BLEND_OP_MULTIPLY_EXT = 1000148012,
    VK_BLEND_OP_SCREEN_EXT = 1000148013,
    VK_BLEND_OP_OVERLAY_EXT = 1000148014,
    VK_BLEND_OP_DARKEN_EXT = 1000148015,
    VK_BLEND_OP_LIGHTEN_EXT = 1000148016,
    VK_BLEND_OP_COLORDODGE_EXT = 1000148017,
} VkBlendOp;
```
VK_BLEND_OP_COLORBURN_EXT = 1000148018,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HARDLIGHT_EXT = 1000148019,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_SOFTLIGHT_EXT = 1000148020,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_DIFFERENCE_EXT = 1000148021,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_EXCLUSION_EXT = 1000148022,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_INVERT_EXT = 1000148023,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_INVERT_RGB_EXT = 1000148024,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LINEARDODGE_EXT = 1000148025,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LINEARBURN_EXT = 1000148026,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_VIVIDLIGHT_EXT = 1000148027,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_LINEARLIGHT_EXT = 1000148028,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PINLIGHT_EXT = 1000148029,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HARDMIX_EXT = 1000148030,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_HUE_EXT = 1000148031,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_SATURATION_EXT = 1000148032,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_COLOR_EXT = 1000148033,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_HSL_LUMINOSITY_EXT = 1000148034,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_EXT = 1000148035,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_CLAMPED_EXT = 1000148036,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT = 1000148037,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_PLUS_DARKER_EXT = 1000148038,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_MINUS_EXT = 1000148039,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_MINUS_CLAMPED_EXT = 1000148040,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_CONTRAST_EXT = 1000148041,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_INVERT_OVG_EXT = 1000148042,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_RED_EXT = 1000148043,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_GREEN_EXT = 1000148044,
// Provided by VK_EXT_blend_operation_advanced
VK_BLEND_OP_BLUE_EXT = 1000148045,
} VkBlendOp;
The semantics of the basic blend operations are described in the table below:

<table>
<thead>
<tr>
<th>VkBlendOp</th>
<th>RGB Components</th>
<th>Alpha Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_ADD</td>
<td>( R = R_{s0} \times S_r + R_{d} \times D_r )</td>
<td>( A = A_{s0} \times S_a + A_{d} \times D_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_{s0} \times S_g + G_{d} \times D_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_{s0} \times S_b + B_{d} \times D_b )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_SUBTRACT</td>
<td>( R = R_{s0} \times S_r - R_{d} \times D_r )</td>
<td>( A = A_{s0} \times S_a - A_{d} \times D_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_{s0} \times S_g - G_{d} \times D_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_{s0} \times S_b - B_{d} \times D_b )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_REVERSE_SUBTRACT</td>
<td>( R = R_{d} \times D_r - R_{s0} \times S_r )</td>
<td>( A = A_{d} \times D_a - A_{s0} \times S_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_{d} \times D_g - G_{s0} \times S_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_{d} \times D_b - B_{s0} \times S_b )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_MIN</td>
<td>( R = \min(R_{s0},R_{d}) )</td>
<td>( A = \min(A_{s0},A_{d}) )</td>
</tr>
<tr>
<td></td>
<td>( G = \min(G_{s0},G_{d}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = \min(B_{s0},B_{d}) )</td>
<td></td>
</tr>
<tr>
<td>VK_BLEND_OP_MAX</td>
<td>( R = \max(R_{s0},R_{d}) )</td>
<td>( A = \max(A_{s0},A_{d}) )</td>
</tr>
<tr>
<td></td>
<td>( G = \max(G_{s0},G_{d}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = \max(B_{s0},B_{d}) )</td>
<td></td>
</tr>
</tbody>
</table>

In this table, the following conventions are used:

- \( R_{s0}, G_{s0}, B_{s0} \) and \( A_{s0} \) represent the first source color R, G, B, and A components, respectively.
- \( R_{d}, G_{d}, B_{d} \) and \( A_{d} \) represent the R, G, B, and A components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- \( S_r, S_g, S_b \) and \( S_a \) represent the source blend factor R, G, B, and A components, respectively.
- \( D_r, D_g, D_b \) and \( D_a \) represent the destination blend factor R, G, B, and A components, respectively.

The blending operation produces a new set of values R, G, B and A, which are written to the framebuffer attachment. If blending is not enabled for this attachment, then R, G, B and A are assigned \( R_{s0}, G_{s0}, B_{s0} \) and \( A_{s0} \), respectively.

If the color attachment is fixed-point, the components of the source and destination values and blend factors are each clamped to \([0,1]\) or \([-1,1]\) respectively for an unsigned normalized or signed normalized color attachment prior to evaluating the blend operations. If the color attachment is floating-point, no clamping occurs.

If the numeric format of a framebuffer attachment uses sRGB encoding, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence are linearized prior to their use in blending. Each R, G, and B component is converted from nonlinear to linear as described in the “sRGB EOTF” section of the Khronos Data Format Specification. If the format is not sRGB, no linearization is performed.

If the numeric format of a framebuffer attachment uses sRGB encoding, then the final R, G and B values are converted into the nonlinear sRGB representation before being written to the framebuffer attachment as described in the “sRGB EOTF” section of the Khronos Data Format Specification.

Chapter 29. The Framebuffer | 1601
If the numeric format of a framebuffer color attachment is not sRGB encoded then the resulting $c$ values for R, G and B are unmodified. The value of A is never sRGB encoded. That is, the alpha component is always stored in memory as linear.

If the framebuffer color attachment is `VK_ATTACHMENT_UNUSED`, no writes are performed through that attachment. Writes are not performed to framebuffer color attachments greater than or equal to the `VkSubpassDescription::colorAttachmentCount` or `VkSubpassDescription2::colorAttachmentCount` value.

### 29.1.4. Advanced Blend Operations

The *advanced blend operations* are those listed in tables *f/X/Y/Z Advanced Blend Operations*, *Hue-Saturation-Luminosity Advanced Blend Operations*, and *Additional RGB Blend Operations*.

If the `pNext` chain of `VkPipelineColorBlendStateCreateInfo` includes a `VkPipelineColorBlendAdvancedStateCreateInfoEXT` structure, then that structure includes parameters that affect advanced blend operations.

The `VkPipelineColorBlendAdvancedStateCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef struct VkPipelineColorBlendAdvancedStateCreateInfoEXT {
    VkStructureType      sType;
    const void*          pNext;
    VkBool32             srcPremultiplied;
    VkBool32             dstPremultiplied;
    VkBlendOverlapEXT    blendOverlap;
} VkPipelineColorBlendAdvancedStateCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `srcPremultiplied` specifies whether the source color of the blend operation is treated as premultiplied.
- `dstPremultiplied` specifies whether the destination color of the blend operation is treated as premultiplied.
- `blendOverlap` is a `VkBlendOverlapEXT` value specifying how the source and destination sample's coverage is correlated.

If this structure is not present, `srcPremultiplied` and `dstPremultiplied` are both considered to be `VK_TRUE`, and `blendOverlap` is considered to be `VK_BLEND_OVERLAP_UNCORRELATED_EXT`. 

Valid Usage

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-srcPremultiplied-01424
  If the non-premultiplied source color property is not supported, **srcPremultiplied** must be **VK_TRUE**

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-dstPremultiplied-01425
  If the non-premultiplied destination color property is not supported, **dstPremultiplied** must be **VK_TRUE**

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-blendOverlap-01426
  If the correlated overlap property is not supported, **blendOverlap** must be **VK_BLEND_OVERLAP_UNCORRELATED_EXT**

Valid Usage (Implicit)

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-sType-sType
  **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT**

- VUID-VkPipelineColorBlendAdvancedStateCreateInfoEXT-blendOverlap-parameter
  **blendOverlap** must be a valid **VkBlendOverlapEXT** value

When using one of the operations in table **f/X/Y/Z Advanced Blend Operations** or **Hue-Saturation-Luminosity Advanced Blend Operations**, blending is performed according to the following equations:

\[
\begin{align*}
R &= f(R_s', R_d') + p_0(A_s, A_d) + Y' \cdot p_1(A_s, A_d) + Z' \cdot p_2(A_s, A_d) \\
G &= f(G_s', G_d') + p_0(A_s, A_d) + Y' \cdot p_1(A_s, A_d) + Z' \cdot p_2(A_s, A_d) \\
B &= f(B_s', B_d') + p_0(A_s, A_d) + Y' \cdot p_1(A_s, A_d) + Z' \cdot p_2(A_s, A_d) \\
A &= X' \cdot p_0(A_s, A_d) + Y' \cdot p_1(A_s, A_d) + Z' \cdot p_2(A_s, A_d)
\end{align*}
\]

where the function **f** and terms **X**, **Y**, and **Z** are specified in the table. The **R**, **G**, and **B** components of the source color used for blending are derived according to **srcPremultiplied**. If **srcPremultiplied** is set to **VK_TRUE**, the fragment color components are considered to have been premultiplied by the **A** component prior to blending. The base source color \((R_s', G_s', B_s')\) is obtained by dividing through by the **A** component:

\[
(R_s', G_s', B_s') = \begin{cases} 
(0, 0, 0) & A_s = 0 \\
\frac{R_s}{A_s}, \frac{G_s}{A_s}, \frac{B_s}{A_s} & \text{otherwise}
\end{cases}
\]

If **srcPremultiplied** is **VK_FALSE**, the fragment color components are used as the base color:

\[
(R_s', G_s', B_s') = (R_s, G_s, B_s)
\]

The **R**, **G**, and **B** components of the destination color used for blending are derived according to **dstPremultiplied**. If **dstPremultiplied** is set to **VK_TRUE**, the destination components are considered to have been premultiplied by the **A** component prior to blending. The base destination color \((R_d', G_d', B_d')\) is obtained by dividing through by the **A** component:
If `dstPremultiplied` is `VK_FALSE`, the destination color components are used as the base color:

\[
(R_d', G_d', B_d') = (R_d, G_d, B_d)
\]

When blending using advanced blend operations, we expect that the R, G, and B components of premultiplied source and destination color inputs be stored as the product of non-premultiplied R, G, and B component values and the A component of the color. If any R, G, or B component of a premultiplied input color is non-zero and the A component is zero, the color is considered ill-formed, and the corresponding component of the blend result is undefined.

All of the advanced blend operation formulas in this chapter compute the result as a premultiplied color. If `dstPremultiplied` is `VK_FALSE`, that result color’s R, G, and B components are divided by the A component before being written to the framebuffer. If any R, G, or B component of the color is non-zero and the A component is zero, the result is considered ill-formed, and the corresponding component of the blend result is undefined. If all components are zero, that value is unchanged.

If the A component of any input or result color is less than zero, the color is considered ill-formed, and all components of the blend result are undefined.

The weighting functions \(p_0\), \(p_1\), and \(p_2\) are defined in table `Advanced Blend Overlap Modes`. In these functions, the A components of the source and destination colors are taken to indicate the portion of the pixel covered by the fragment (source) and the fragments previously accumulated in the pixel (destination). The functions \(p_0\), \(p_1\), and \(p_2\) approximate the relative portion of the pixel covered by the intersection of the source and destination, covered only by the source, and covered only by the destination, respectively.

Possible values of `VkPipelineColorBlendAdvancedStateCreateInfoEXT::blendOverlap`, specifying the blend overlap functions, are:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef enum VkBlendOverlapEXT {
    VK_BLEND_OVERLAP_UNCORRELATED_EXT = 0,
    VK_BLEND_OVERLAP_DISJOINT_EXT = 1,
    VK_BLEND_OVERLAP_CONJOINT_EXT = 2,
} VkBlendOverlapEXT;
```

- `VK_BLEND_OVERLAP_UNCORRELATED_EXT` specifies that there is no correlation between the source and destination coverage.
- `VK_BLEND_OVERLAP_CONJOINT_EXT` specifies that the source and destination coverage are considered to have maximal overlap.
- `VK_BLEND_OVERLAP_DISJOINT_EXT` specifies that the source and destination coverage are considered to have minimal overlap.

*Table 40. Advanced Blend Overlap Modes*
### Table 41. f/X/Y/Z Advanced Blend Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>Blend Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OVERLAP_UNCORRELATED_EXT</td>
<td>( p_0(A_s, A_d) = A_s A_d ) \hspace{1cm} ( p_1(A_s, A_d) = A_s(1 - A_d) ) \hspace{1cm} ( p_2(A_s, A_d) = A_d(1 - A_s) )</td>
</tr>
<tr>
<td>VK_BLEND_OVERLAP_CONJOINT_EXT</td>
<td>( p_0(A_s, A_d) = \min(A_s, A_d) ) \hspace{1cm} ( p_1(A_s, A_d) = \max(A_s - A_d, 0) ) \hspace{1cm} ( p_2(A_s, A_d) = \max(A_d - A_s, 0) )</td>
</tr>
<tr>
<td>VK_BLEND_OVERLAP_DISJOINT_EXT</td>
<td>( p_0(A_s, A_d) = \max(A_s + A_d - 1, 0) ) \hspace{1cm} ( p_1(A_s, A_d) = \min(A_s, 1 - A_d) ) \hspace{1cm} ( p_2(A_s, A_d) = \min(A_d, 1 - A_s) )</td>
</tr>
</tbody>
</table>

\[\begin{align*}
(X, Y, Z) & = (0, 0, 0) \\
f(C_s, C_d) & = 0
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 1, 0) \\
f(C_s, C_d) & = C_s
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 0, 1) \\
f(C_s, C_d) & = C_d
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 1, 1) \\
f(C_s, C_d) & = C_s
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 1, 1) \\
f(C_s, C_d) & = C_d
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 0, 0) \\
f(C_s, C_d) & = C_s
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 0, 0) \\
f(C_s, C_d) & = C_d
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (0, 1, 0) \\
f(C_s, C_d) & = 0
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (0, 0, 1) \\
f(C_s, C_d) & = 0
\end{align*}\]

\[\begin{align*}
(X, Y, Z) & = (1, 0, 1) \\
f(C_s, C_d) & = C_s
\end{align*}\]
<table>
<thead>
<tr>
<th>Mode</th>
<th>Blend Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VK_BLEND_OP_DST_ATOP_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 0))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = C_d)</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_XOR_EXT</strong></td>
<td>((X, Y, Z) = (0, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = 0)</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_MULTIPLY_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = C_sC_d)</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_SCREEN_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = C_s + C_d - C_sC_d)</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_OVERLAY_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \begin{cases} 2C_sC_d &amp; C_d \leq 0 \ 1 - 2(1 - C_s)(1 - C_d) &amp; \text{otherwise} \end{cases})</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_DARKEN_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \min(C_s, C_d))</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_LIGHTEN_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \max(C_s, C_d))</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_COLORDODGE_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \begin{cases} 0 &amp; C_d \leq 0 \ \min(1, \frac{C_d}{1 - C_s}) &amp; C_d &gt; 0 \text{and} C_s &lt; 1 \ 1 &amp; C_d &gt; 0 \text{and} C_s \geq 1 \end{cases})</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_COLORBURN_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \begin{cases} 1 &amp; C_d \geq 1 \ 1 - \min(1, \frac{1 - C_d}{C_s}) &amp; C_d &lt; 1 \text{and} C_s &gt; 0 \ 0 &amp; C_d &lt; 1 \text{and} C_s \leq 0 \end{cases})</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_HARDLIGHT_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \begin{cases} 2C_sC_d &amp; C_s \leq 0.5 \ 1 - 2(1 - C_s)(1 - C_d) &amp; \text{otherwise} \end{cases})</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_SOFTLIGHT_EXT</strong></td>
<td>((X, Y, Z) = (3, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = \begin{cases} C_s - (1 - 2C_d)C_d - C_d &amp; C_s \leq 0.5 \ C_s \times (2C_d - 1)(12C_d + 3) &amp; C_s &gt; 0.5 \text{and} C_s \leq 0.25 \ C_s \times (2C_d - 1)(12C_d + 3) &amp; C_s &gt; 0.5 \text{and} C_s &gt; 0.25 \end{cases})</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_DIFFERENCE_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) =</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_EXCLUSION_EXT</strong></td>
<td>((X, Y, Z) = (1, 1, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = C_s + C_d - 2C_sC_d)</td>
</tr>
<tr>
<td><strong>VK_BLEND_OP_INVERT_EXT</strong></td>
<td>((X, Y, Z) = (1, 0, 1))</td>
</tr>
<tr>
<td></td>
<td>(f(C_s, C_d) = 1 - C_d)</td>
</tr>
<tr>
<td>Mode</td>
<td>Blend Coefficients</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VK_BLEND_OP_INVERT_RGB_EXT</td>
<td>( (X, Y, Z) = (1, 0, 1) )  ( f(C_s, C_d) = C_s(1 - C_d) )</td>
</tr>
</tbody>
</table>
| VK_BLEND_OP_LINEARDODGE_EXT              | \( (X, Y, Z) = (1, 1, 1) \)  \( f(C_s, C_d) = \begin{cases} 
C_s + C_d & \text{if } C_s + C_d \leq 1 \\
1 & \text{otherwise} \end{cases} \) |
| VK_BLEND_OP_LINEARBURN_EXT               | \( (X, Y, Z) = (1, 1, 1) \)  \( f(C_s, C_d) = \begin{cases} 
C_s + C_d - 1 & \text{if } C_s + C_d > 1 \\
0 & \text{otherwise} \end{cases} \) |
| VK_BLEND_OP_VIVIDLIGHT_EXT               | \( (X, Y, Z) = (1, 1, 1) \)  \( f(C_s, C_d) = \begin{cases} 
1 - \min(1, \frac{1 - C_d}{2C_s}) & 0 < C_s < 0.5 \\
0 & C_s \leq 0 \\
\min(1, \frac{C_d}{2(1 - C_s)}) & 0.5 \leq C_s < 1 \\
1 & C_s \geq 1 \end{cases} \) |
| VK_BLEND_OP_LINEARLIGHT_EXT              | \( (X, Y, Z) = (1, 1, 1) \)  \( f(C_s, C_d) = \begin{cases} 
1 & 2C_s + C_d > 2 \\
2C_s + C_d - 1 & 1 < 2C_s + C_d \leq 2 \\
0 & 2C_s + C_d \leq 1 \end{cases} \) |
| VK_BLEND_OP_PINLIGHT_EXT                 | \( (X, Y, Z) = (1, 1, 1) \)  \( f(C_s, C_d) = \begin{cases} 
0 & 2C_s - 1 > 0 \text{and} C_s \leq 0.5 \\
2C_s - 1 & 2C_s - 1 > 0 \text{and} C_s > 0.5 \\
C_d & 2C_s - 1 \leq 0 \text{and} C_s \leq 0.5 \text{and} C_d \leq 0.5 \end{cases} \) |
| VK_BLEND_OP_HARDMIX_EXT                  | \( (X, Y, Z) = (1, 1, 1) \)  \( f(C_s, C_d) = \begin{cases} 
0 & C_s + C_d < 1 \\
1 & \text{otherwise} \end{cases} \) |

When using one of the HSL blend operations in table **Hue-Saturation-Luminosity Advanced Blend Operations** as the blend operation, the RGB color components produced by the function \( f \) are effectively obtained by converting both the non-premultiplied source and destination colors to the HSL (hue, saturation, luminosity) color space, generating a new HSL color by selecting H, S, and L components from the source or destination according to the blend operation, and then converting the result back to RGB. In the equations below, a blended RGB color is produced according to the following pseudocode:

```c
float minv3(vec3 c) {
    return min(min(c.r, c.g), c.b);
}
float maxv3(vec3 c) {
    return max(max(c.r, c.g), c.b);
}
float lumv3(vec3 c) {
```
\[ \text{dot(c, vec3(0.30, 0.59, 0.11))} \]
}

\begin{verbatim}
float satv3(vec3 c) {
    return maxv3(c) - minv3(c);
}

// If any color components are outside [0,1], adjust the color to
// get the components in range.
vec3 ClipColor(vec3 color) {
    float lum = lumv3(color);
    float mincol = minv3(color);
    float maxcol = maxv3(color);
    if (mincol < 0.0) {
        color = lum + ((color-lum)*lum) / (lum-mincol);
    }
    if (maxcol > 1.0) {
        color = lum + ((color-lum)*(1-lum)) / (maxcol-lum);
    }
    return color;
}

// Take the base RGB color <cbase> and override its luminosity
// with that of the RGB color <clum>.
vec3 SetLum(vec3 cbase, vec3 clum) {
    float lbase = lumv3(cbase);
    float llum = lumv3(clum);
    float ldiff = llum - lbase;
    vec3 color = cbase + vec3(ldiff);
    return ClipColor(color);
}

// Take the base RGB color <cbase> and override its saturation with
// that of the RGB color <csat>. The override the luminosity of the
// result with that of the RGB color <clum>.
vec3 SetLumSat(vec3 cbase, vec3 csat, vec3 clum) {
    float minbase = minv3(cbase);
    float sbase = satv3(cbase);
    float ssat = satv3(csat);
    vec3 color;
    if (sbase > 0) {
        // Equivalent (modulo rounding errors) to setting the
        // smallest (R,G,B) component to 0, the largest to <ssat>,
        // and interpolating the "middle" component based on its
        // original value relative to the smallest/largest.
        color = (cbase - minbase) * ssat / sbase;
    } else {
        color = vec3(0.0);
    }
    return SetLum(color, clum);
}
\end{verbatim}
Table 42. Hue-Saturation-Luminosity Advanced Blend Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>Result</th>
</tr>
</thead>
</table>
| VK_BLEND_OP_HSL_HUE_EXT               | \((X, Y, Z) = (1, 1, 1)\)  
\(f(C_s, C_d) = SetLumSat(C_s, C_d)\) |
| VK_BLEND_OP_HSL_SATURATION_EXT        | \((X, Y, Z) = (1, 1, 1)\)  
\(f(C_s, C_d) = SetLumSat(C_d, C_s, C_d)\) |
| VK_BLEND_OP_HSL_COLOR_EXT             | \((X, Y, Z) = (1, 1, 1)\)  
\(f(C_s, C_d) = SetLum(C_s, C_d)\) |
| VK_BLEND_OP_HSL_LUMINOSITY_EXT        | \((X, Y, Z) = (1, 1, 1)\)  
\(f(C_s, C_d) = SetLum(C_d, C_s)\) |

When using one of the operations in table Additional RGB Blend Operations as the blend operation, the source and destination colors used by these blending operations are interpreted according to srcPremultiplied and dstPremultiplied. The blending operations below are evaluated where the RGB source and destination color components are both considered to have been premultiplied by the corresponding A component.

\[
(R_s, G_s, B_s) = \begin{cases} 
(R_s, G_s, B_s) & \text{if srcPremultiplied is VK_TRUE} \\
(R_sA_s, G_sA_s, B_sA_s) & \text{if srcPremultiplied is VK_FALSE}
\end{cases}
\]

\[
(R_d, G_d, B_d) = \begin{cases} 
(R_d, G_d, B_d) & \text{if dstPremultiplied is VK_TRUE} \\
(R_dA_d, G_dA_d, B_dA_d) & \text{if dstPremultiplied is VK_FALSE}
\end{cases}
\]

Table 43. Additional RGB Blend Operations

<table>
<thead>
<tr>
<th>Mode</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_PLUS_EXT</td>
<td>((R, G, B, A) = (R_s + R_d, G_s + G_d, B_s + B_d, A_s + A_d))</td>
</tr>
<tr>
<td>VK_BLEND_OP_PLUS_CLAMPED_EXT</td>
<td>((R, G, B, A) = (\min(1, R_s + R_d), \min(1, G_s + G_d), \min(1, B_s + B_d), \min(1, A_s + A_d)))</td>
</tr>
<tr>
<td>VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT</td>
<td>((R, G, B, A) = (\min(\min(1, A_s + A_d), R_s + R_d), \min(\min(1, A_s + A_d), G_s + G_d), \min(\min(1, A_s + A_d), B_s + B_d), \min(1, A_s + A_d)))</td>
</tr>
<tr>
<td>VK_BLEND_OP_PLUS_DARKER_EXT</td>
<td>((R, G, B, A) = (\max(0, \min(1, A_s + A_d) - ((A_s - R_s) + (A_d - R_d))), \max(0, \min(1, A_s + A_d) - ((A_s - G_s) + (A_d - G_d))), \max(0, \min(1, A_s + A_d) - ((A_s - B_s) + (A_d - B_d))), \min(1, A_s + A_d)))</td>
</tr>
</tbody>
</table>
### 29.2. Logical Operations

The application can enable a logical operation between the fragment's color values and the existing value in the framebuffer attachment. This logical operation is applied prior to updating the framebuffer attachment. Logical operations are applied only for signed and unsigned integer and normalized integer framebuffers. Logical operations are not applied to floating-point or sRGB format color attachments.

Logical operations are controlled by the `logicOpEnable` and `logicOp` members of `VkPipelineColorBlendStateCreateInfo`. It can also be controlled by `vkCmdSetLogicOpEXT` if graphics pipeline is created with `VK_DYNAMIC_STATE_LOGIC_OP_EXT` set in `VkPipelineDynamicStateCreateInfo`::pDynamicStates. If `logicOpEnable` is `VK_TRUE`, then a logical operation selected by `logicOp` is applied between each color attachment and the fragment's corresponding output value, and blending of all attachments is treated as if it were disabled. Any attachments using color formats for which logical operations are not supported simply pass through the color values unmodified. The logical operation is applied independently for each of the red, green, blue, and alpha components. The `logicOp` is selected from the following operations:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_BLEND_OP_MINUS_EXT</td>
<td>[(R, G, B, A) = (R_d' - R_s', G_d' - G_s', B_d' - B_s', A_d - A_s)]</td>
</tr>
<tr>
<td>VK_BLEND_OP_MINUS_CLAMPED_EXT</td>
<td>[(R, G, B, A) = (\max(0, R_d' - R_s'), \max(0, G_d' - G_s'), \max(0, B_d' - B_s'), \max(0, A_d - A_s))]</td>
</tr>
<tr>
<td>VK_BLEND_OP_CONTRAST_EXT</td>
<td>[(R, G, B, A) = \left(\frac{A_d}{2} + 2R_d' - \frac{A_d}{2} \cdot \frac{R_s}{2}, \frac{A_d}{2} + 2G_d' - \frac{A_d}{2} \cdot \frac{G_s}{2}, \frac{A_d}{2} + 2B_d' - \frac{A_d}{2} \cdot \frac{B_s}{2}, \frac{A_d}{2}\right)]</td>
</tr>
<tr>
<td>VK_BLEND_OP_RED_EXT</td>
<td>[(R, G, B, A) = (R_s', G_d', B_d', A_d)]</td>
</tr>
<tr>
<td>VK_BLEND_OP_GREEN_EXT</td>
<td>[(R, G, B, A) = (R_d', G_s', B_d', A_d)]</td>
</tr>
<tr>
<td>VK_BLEND_OP_BLUE_EXT</td>
<td>[(R, G, B, A) = (R_d', G_d', B_s', A_d)]</td>
</tr>
</tbody>
</table>
// Provided by VK_VERSION_1_0

typedef enum VkLogicOp {
    VK_LOGIC_OP_CLEAR = 0,
    VK_LOGIC_OP_AND = 1,
    VK_LOGIC_OP_AND_REVERSE = 2,
    VK_LOGIC_OP_COPY = 3,
    VK_LOGIC_OP_AND_INVERTED = 4,
    VK_LOGIC_OP_NO_OP = 5,
    VK_LOGIC_OP_XOR = 6,
    VK_LOGIC_OP_OR = 7,
    VK_LOGIC_OP_NOR = 8,
    VK_LOGIC_OP_EQUIVALENT = 9,
    VK_LOGIC_OP_INVERT = 10,
    VK_LOGIC_OP_OR_REVERSE = 11,
    VK_LOGIC_OP_COPY_INVERTED = 12,
    VK_LOGIC_OP_OR_INVERTED = 13,
    VK_LOGIC_OP_NAND = 14,
    VK_LOGIC_OP_SET = 15,
} VkLogicOp;
The logical operations supported by Vulkan are summarized in the following table in which

- \( \neg \) is bitwise invert,
- \( \land \) is bitwise and,
- \( \lor \) is bitwise or,
- \( \oplus \) is bitwise exclusive or,
- \( s \) is the fragment's \( R_{s0}, G_{s0}, B_{s0} \) or \( A_{s0} \) component value for the fragment output corresponding to the color attachment being updated, and
- \( d \) is the color attachment's \( R, G, B \) or \( A \) component value:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_LOGIC_OP_CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND</td>
<td>( s \land d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_REVERSE</td>
<td>( s \land \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY</td>
<td>( s )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_INVERTED</td>
<td>( \neg s \land d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NO_OP</td>
<td>( d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_XOR</td>
<td>( s \oplus d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR</td>
<td>( s \lor d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NOR</td>
<td>( \neg (s \lor d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_EQUIVALENT</td>
<td>( \neg (s \oplus d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_INVERT</td>
<td>( \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_REVERSE</td>
<td>( s \lor \neg d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY_INVERTED</td>
<td>( \neg s )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_INVERTED</td>
<td>( \neg s \lor d )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NAND</td>
<td>( \neg (s \land d) )</td>
</tr>
<tr>
<td>VK_LOGIC_OP_SET</td>
<td>all 1s</td>
</tr>
</tbody>
</table>

The result of the logical operation is then written to the color attachment as controlled by the component write mask, described in Blend Operations.

To dynamically set the logical operation to apply for blend state:

```c
// Provided by VK_EXTExtendedDynamicState2
void vkCmdSetLogicOpEXT(  
    VkCommandBuffer commandBuffer,  
    VkLogicOp logicOp);
```
• **commandBuffer** is the command buffer into which the command will be recorded.
• **logicOp** specifies the logical operation to apply for blend state.

This command sets the state for a given draw when the graphics pipeline is created with **VK_DYNAMIC_STATE_LOGIC_OP_EXT** set in **VkPipelineDynamicStateCreateInfo::pDynamicStates**.

### Valid Usage

- **VUID-vkCmdSetLogicOpEXT-None-04867**
  - The **extendedDynamicState2LogicOp** feature **must** be enabled

### Valid Usage (Implicit)

- **VUID-vkCmdSetLogicOpEXT-commandBuffer-parameter**
  - **commandBuffer** **must** be a valid **VkCommandBuffer** handle
- **VUID-vkCmdSetLogicOpEXT-logicOp-parameter**
  - **logicOp** **must** be a valid **VkLogicOp** value
- **VUID-vkCmdSetLogicOpEXT-commandBuffer-recording**
  - **commandBuffer** **must** be in the **recording state**
- **VUID-vkCmdSetLogicOpEXT-commandBuffer-cmdpool**
  - The **VkCommandPool** that **commandBuffer** was allocated from **must** support graphics operations

### Host Synchronization

- Host access to **commandBuffer** **must** be externally synchronized
- Host access to the **VkCommandPool** that **commandBuffer** was allocated from **must** be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 29.3. Color Write Mask

Bits which **can** be set in **VkPipelineColorBlendAttachmentState::colorWriteMask** to determine whether the final color values R, G, B and A are written to the framebuffer attachment are:
typedef enum VkColorComponentFlagBits {
    VK_COLOR_COMPONENT_R_BIT = 0x00000001,
    VK_COLOR_COMPONENT_G_BIT = 0x00000002,
    VK_COLOR_COMPONENT_B_BIT = 0x00000004,
    VK_COLOR_COMPONENT_A_BIT = 0x00000008,
} VkColorComponentFlagBits;

- **VK_COLOR_COMPONENT_R_BIT** specifies that the R value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK_COLOR_COMPONENT_G_BIT** specifies that the G value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK_COLOR_COMPONENT_B_BIT** specifies that the B value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.
- **VK_COLOR_COMPONENT_A_BIT** specifies that the A value is written to the color attachment for the appropriate sample. Otherwise, the value in memory is unmodified.

The color write mask operation is applied regardless of whether blending is enabled.

The color write mask operation is applied only if **Color Write Enable** is enabled for the respective attachment. Otherwise the color write mask is ignored and writes to all components of the attachment are disabled.

typedef VkFlags VkColorComponentFlags;

VkColorComponentFlags is a bitmask type for setting a mask of zero or more VkColorComponentFlagBits.

### 29.4. Color Write Enable

The **VkPipelineColorWriteCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_color_write_enable
typedef struct VkPipelineColorWriteCreateInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           attachmentCount;
    const VkBool32*    pColorWriteEnables;
} VkPipelineColorWriteCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **attachmentCount** is the number of **VkBool32** elements in **pColorWriteEnables**.
• `pColorWriteEnables` is a pointer to an array of per target attachment boolean values specifying whether color writes are enabled for the given attachment.

When this structure is included in the `pNext` chain of `VkPipelineColorBlendStateCreateInfo`, it defines per-attachment color write state. If this structure is not included in the `pNext` chain, it is equivalent to specifying this structure with `attachmentCount` equal to the `attachmentCount` member of `VkPipelineColorBlendStateCreateInfo`, and `pColorWriteEnables` set to an array of as many `VK_TRUE` values.

If the `colorWriteEnable` feature is not enabled on the device, all `VkBool32` elements in the `pColorWriteEnables` array must be `VK_TRUE`.

Color Write Enable interacts with the `Color Write Mask` as follows: * If `colorWriteEnable` is `VK_TRUE`, writes to the attachment are determined by the `colorWriteMask`. * If `colorWriteEnable` is `VK_FALSE`, the `colorWriteMask` is ignored and writes to all components of the attachment are disabled. This is equivalent to specifying a `colorWriteMask` of 0.

### Valid Usage

- VUID-VkPipelineColorWriteCreateInfoEXT-pAttachments-04801
  
  If the `colorWriteEnable` feature is not enabled, all elements of `pColorWriteEnables` must be `VK_TRUE`

- VUID-VkPipelineColorWriteCreateInfoEXT-attachmentCount-04802
  
  `attachmentCount` must be equal to the `attachmentCount` member of the `VkPipelineColorBlendStateCreateInfo` structure specified during pipeline creation

### Valid Usage (Implicit)

- VUID-VkPipelineColorWriteCreateInfoEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_COLOR_WRITE_CREATE_INFO_EXT`

- VUID-VkPipelineColorWriteCreateInfoEXT-pColorWriteEnables-parameter
  
  If `attachmentCount` is not 0, `pColorWriteEnables` must be a valid pointer to an array of `attachmentCount` `VkBool32` values

To dynamically enable or disable color writes, call:

```
// Provided by VK_EXT_color_write_enable
void vkCmdSetColorWriteEnableEXT(VkCommandBuffer commandBuffer, uint32_t attachmentCount, const VkBool32* pColorWriteEnables);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `attachmentCount` is the number of `VkBool32` elements in `pColorWriteEnables`.
• `pColorWriteEnables` is a pointer to an array of per target attachment boolean values specifying whether color writes are enabled for the given attachment.

This command sets the state for a given draw when the graphics pipeline is created with `VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

---

**Valid Usage**

- VUID-vkCmdSetColorWriteEnableEXT-None-04803
  The `colorWriteEnable` feature must be enabled

- VUID-vkCmdSetColorWriteEnableEXT-attachmentCount-04804
  `attachmentCount` must be equal to the `attachmentCount` member of the `VkPipelineColorBlendStateCreateInfo` structure specified during pipeline creation

---

**Valid Usage (Implicit)**

- VUID-vkCmdSetColorWriteEnableEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetColorWriteEnableEXT-pColorWriteEnables-parameter
  `pColorWriteEnables` must be a valid pointer to an array of `attachmentCount` `VkBool32` values

- VUID-vkCmdSetColorWriteEnableEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetColorWriteEnableEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations

- VUID-vkCmdSetColorWriteEnableEXT-attachmentCount-arraylength
  `attachmentCount` must be greater than 0

---

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

---

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 30. Dispatching Commands

Dispatching commands (commands with `Dispatch` in the name) provoke work in a compute pipeline. Dispatching commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the bound compute pipeline. A compute pipeline must be bound to a command buffer before any dispatch commands are recorded in that command buffer.

To record a dispatch, call:

```c
// Provided by VK_VERSION_1_0
void vkCmdDispatch(
    VkCommandBuffer commandBuffer,
    uint32_t groupCountX,
    uint32_t groupCountY,
    uint32_t groupCountZ);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `groupCountX` is the number of local workgroups to dispatch in the X dimension.
- `groupCountY` is the number of local workgroups to dispatch in the Y dimension.
- `groupCountZ` is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of `groupCountX \times groupCountY \times groupCountZ` local workgroups is assembled.
Valid Usage

- **VUID-vkCmdDispatch-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatch-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdDispatch-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdDispatch-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- **VUID-vkCmdDispatch-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDispatch-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdDispatch-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- **VUID-vkCmdDispatch-None-02697**
  For each set $n$ that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to $n$ at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set $n$, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in `Pipeline Layout Compatibility`

- **VUID-vkCmdDispatch-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in `Pipeline Layout Compatibility`
Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a ` VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If `commandBuffer` is an unprotected command buffer, any resource accessed by the
- If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction must have at least as many components as the image view’s format.

- If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the `Type` of the `Texel` operand of that instruction must have at least as many components as the buffer view’s format.

- If a `VkImageView` with a `VkFormat` that has a 64-bit channel width is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction must have a `Width` of 64.

- If a `VkBufferView` with a `VkFormat` that has a channel width less than 64-bit is accessed as a result of this command, the `SampledType` of the `OpTypeImage` operand of that instruction must have a `Width` of 32.

- If the `sparseImageInt64Atomics` feature is not enabled, `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

- If the `sparseImageInt64Atomics` feature is not enabled, `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a `SampledType` with a `Width` of 64 by this command.

- If `commandBuffer` is a protected command buffer, any resource written to by the `VkPipeline` object bound to the pipeline bind point used by this command must not be an unprotected resource.

- If `commandBuffer` is a protected command buffer, pipeline stages other than the framebuffer-space and compute stages in the `VkPipeline` object bound to the pipeline bind point used by this command must not write to any resource.
If any of the shader stages of the VkPipeline bound to the pipeline bind point used by this command uses the RayQueryKHR capability, then commandBuffer must not be a protected command buffer.

- VUID-vkCmdDispatch-groupCountX-00386
  `groupCountX` must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]

- VUID-vkCmdDispatch-groupCountY-00387
  `groupCountY` must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]

- VUID-vkCmdDispatch-groupCountZ-00388
  `groupCountZ` must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]

Valid Usage (Implicit)

- VUID-vkCmdDispatch-commandBuffer-parameter
  `commandBuffer` must be a valid VkCommandBuffer handle

- VUID-vkCmdDispatch-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdDispatch-commandBuffer-cmdpool
  The VkCommandPool that `commandBuffer` was allocated from must support compute operations

- VUID-vkCmdDispatch-renderpass
  This command must only be called outside of a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the VkCommandPool that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary, Secondary</td>
<td>Outside</td>
<td>Compute</td>
<td>Compute</td>
</tr>
</tbody>
</table>

To record an indirect command dispatch, call:
void vkCmdDispatchIndirect(
    VkCommandBuffer commandBuffer,
    VkBuffer buffer,
    VkDeviceSize offset);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `buffer` is the buffer containing dispatch parameters.
- `offset` is the byte offset into `buffer` where parameters begin.

`vkCmdDispatchIndirect` behaves similarly to `vkCmdDispatch` except that the parameters are read by the device from a buffer during execution. The parameters of the dispatch are encoded in a `VkDispatchIndirectCommand` structure taken from `buffer` starting at `offset`. 
Valid Usage

- VUID-vkCmdDispatchIndirect-magFilter-04553
  If a VkSampler created with magFilter or minFilter equal to VK_FILTER_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-vkCmdDispatchIndirect-mipmapMode-04770
  If a VkSampler created with mipmapMode equal to VK_SAMPLER_MIPMAP_MODE_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

- VUID-vkCmdDispatchIndirect-None-02691
  If a VkImageView is accessed using atomic operations as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT

- VUID-vkCmdDispatchIndirect-None-02692
  If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

- VUID-vkCmdDispatchIndirect-filterCubic-02694
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2

- VUID-vkCmdDispatchIndirect-filterCubicMinmax-02695
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

- VUID-vkCmdDispatchIndirect-flags-02696
  Any VkImage created with a VkImageCreateInfo::flags containing VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV sampled as a result of this command must only be sampled using a VkSamplerAddressMode of VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE

- VUID-vkCmdDispatchIndirect-None-02697
  For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

- VUID-vkCmdDispatchIndirect-None-02698
  For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility
Compatibility

- VUID-vkCmdDispatchIndirect-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdDispatchIndirect-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdDispatchIndirect-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdDispatchIndirect-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdDispatchIndirect-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdDispatchIndirect-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdDispatchIndirect-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdDispatchIndirect-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDispatchIndirect-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdDispatchIndirect-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
A **VkPipeline** object bound to the pipeline bind point used by this command **must** not be a protected resource.

- **VUID-vkCmdDispatchIndirect-None-04115**
  If a **VkImageView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction **must** have at least as many components as the image view’s format.

- **VUID-vkCmdDispatchIndirect-OpImageWrite-04469**
  If a **VkBufferView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction **must** have at least as many components as the buffer view’s format.

- **VUID-vkCmdDispatchIndirect-SampledType-04470**
  If a **VkImageView** with a **VkFormat** that has a 64-bit channel width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 64.

- **VUID-vkCmdDispatchIndirect-SampledType-04471**
  If a **VkImageView** with a **VkFormat** that has a channel width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 32.

- **VUID-vkCmdDispatchIndirect-SampledType-04472**
  If a **VkBufferView** with a **VkFormat** that has a 64-bit channel width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 64.

- **VUID-vkCmdDispatchIndirect-SampledType-04473**
  If a **VkBufferView** with a **VkFormat** that has a channel width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction **must** have a **Width** of 32.

- **VUID-vkCmdDispatchIndirect-sparseImageInt64Atomics-04474**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkImage** objects created with the **VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT** flag **must** not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdDispatchIndirect-sparseImageInt64Atomics-04475**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkBuffer** objects created with the **VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT** flag **must** not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdDispatchIndirect-buffer-02708**
  If **buffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object.

- **VUID-vkCmdDispatchIndirect-buffer-02709**
  **buffer** **must** have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set.

- **VUID-vkCmdDispatchIndirect-offset-02710**
  **offset** **must** be a multiple of 4.

- **VUID-vkCmdDispatchIndirect-commandBuffer-02711**
  **commandBuffer** **must** not be a protected command buffer.
The sum of `offset` and the size of `VkDispatchIndirectCommand` must be less than or equal to the size of `buffer`.

### Valid Usage (Implicit)

- VUID-vkCmdDispatchIndirect-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdDispatchIndirect-buffer-parameter
  
  `buffer` must be a valid `VkBuffer` handle

- VUID-vkCmdDispatchIndirect-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdDispatchIndirect-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations

- VUID-vkCmdDispatchIndirect-renderpass
  
  This command must only be called outside of a render pass instance

- VUID-vkCmdDispatchIndirect-commonparent
  
  Both of `buffer`, and `commandBuffer` must have been created, allocated, or retrieved from the same `VkDevice`

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td>Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkDispatchIndirectCommand` structure is defined as:
typedef struct VkDispatchIndirectCommand {
    uint32_t    x;
    uint32_t    y;
    uint32_t    z;
} VkDispatchIndirectCommand;

- \( x \) is the number of local workgroups to dispatch in the X dimension.
- \( y \) is the number of local workgroups to dispatch in the Y dimension.
- \( z \) is the number of local workgroups to dispatch in the Z dimension.

The members of VkDispatchIndirectCommand have the same meaning as the corresponding parameters of vkCmdDispatch.

**Valid Usage**

- VUID-VkDispatchIndirectCommand-x-00417
  \( x \) must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]

- VUID-VkDispatchIndirectCommand-y-00418
  \( y \) must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]

- VUID-VkDispatchIndirectCommand-z-00419
  \( z \) must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]

To record a dispatch using non-zero base values for the components of WorkgroupId, call:

```c
// Provided by VK_VERSION_1_1
void vkCmdDispatchBase(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    baseGroupX,
    uint32_t                                    baseGroupY,
    uint32_t                                    baseGroupZ,
    uint32_t                                    groupCountX,
    uint32_t                                    groupCountY,
    uint32_t                                    groupCountZ);
```

or the equivalent command
```c
// Provided by VK_KHR_device_group
void vkCmdDispatchBaseKHR(
    VkCommandBuffer commandBuffer,
    uint32_t baseGroupX,  // is the start value for the X component of WorkgroupId.
    uint32_t baseGroupY,  // is the start value for the Y component of WorkgroupId.
    uint32_t baseGroupZ,  // is the start value for the Z component of WorkgroupId.
    uint32_t groupCountX, // is the number of local workgroups to dispatch in the X dimension.
    uint32_t groupCountY, // is the number of local workgroups to dispatch in the Y dimension.
    uint32_t groupCountZ); // is the number of local workgroups to dispatch in the Z dimension.
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `baseGroupX` is the start value for the X component of `WorkgroupId`.
- `baseGroupY` is the start value for the Y component of `WorkgroupId`.
- `baseGroupZ` is the start value for the Z component of `WorkgroupId`.
- `groupCountX` is the number of local workgroups to dispatch in the X dimension.
- `groupCountY` is the number of local workgroups to dispatch in the Y dimension.
- `groupCountZ` is the number of local workgroups to dispatch in the Z dimension.

When the command is executed, a global workgroup consisting of `groupCountX × groupCountY × groupCountZ` local workgroups is assembled, with `WorkgroupId` values ranging from `[baseGroup*, baseGroup* + groupCount*) in each component. `vkCmdDispatch` is equivalent to `vkCmdDispatchBase(0, 0, 0, groupCountX, groupCountY, groupCountZ).`
Valid Usage

• VUID-vkCmdDispatchBase-magFilter-04553
  If a VkSampler created with magFilter or minFilter equal to VK_FILTER_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view’s format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

• VUID-vkCmdDispatchBase-mipmapMode-04770
  If a VkSampler created with mipmapMode equal to VK_SAMPLER_MIPMAP_MODE_LINEAR and compareEnable equal to VK_FALSE is used to sample a VkImageView as a result of this command, then the image view’s format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT

• VUID-vkCmdDispatchBase-None-02691
  If a VkImageView is accessed using atomic operations as a result of this command, then the image view’s format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT

• VUID-vkCmdDispatchBase-None-02692
  If a VkImageView is sampled with VK_FILTER_CUBIC_EXT as a result of this command, then the image view’s format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT

• VUID-vkCmdDispatchBase-filterCubic-02694
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT as a result of this command must have a VkImageViewType and format that supports cubic filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic returned by vkGetPhysicalDeviceImageFormatProperties2

• VUID-vkCmdDispatchBase-filterCubicMinmax-02695
  Any VkImageView being sampled with VK_FILTER_CUBIC_EXT with a reduction mode of either VK_SAMPLER_REDUCTION_MODE_MIN or VK_SAMPLER_REDUCTION_MODE_MAX as a result of this command must have a VkImageViewType and format that supports cubic filtering together with minmax filtering, as specified by VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax returned by vkGetPhysicalDeviceImageFormatProperties2

• VUID-vkCmdDispatchBase-flags-02696
  Any VkImage created with a VkImageCreateInfo::flags containing VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV sampled as a result of this command must only be sampled using a VkSamplerAddressMode of VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE

• VUID-vkCmdDispatchBase-None-02697
  For each set n that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a VkPipelineLayout that is compatible for set n, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout Compatibility

• VUID-vkCmdDispatchBase-None-02698
  For each push constant that is statically used by the VkPipeline bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Pipeline Layout
Compatibility

- **VUID-vkCmdDispatchBase-None-02699**
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- **VUID-vkCmdDispatchBase-None-02700**
  A valid pipeline must be bound to the pipeline bind point used by this command.

- **VUID-vkCmdDispatchBase-commandBuffer-02701**
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- **VUID-vkCmdDispatchBase-None-02859**
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- **VUID-vkCmdDispatchBase-None-02702**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- **VUID-vkCmdDispatchBase-None-02703**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- **VUID-vkCmdDispatchBase-None-02704**
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- **VUID-vkCmdDispatchBase-None-02705**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDispatchBase-None-02706**
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- **VUID-vkCmdDispatchBase-commandBuffer-02707**
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource.

- VUID-vkCmdDispatchBase-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- VUID-vkCmdDispatchBase-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- VUID-vkCmdDispatchBase-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDispatchBase-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDispatchBase-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdDispatchBase-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdDispatchBase-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDispatchBase-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdDispatchBase-baseGroupX-00421
  baseGroupX must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]

- VUID-vkCmdDispatchBase-baseGroupX-00422
  baseGroupY must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]

- VUID-vkCmdDispatchBase-baseGroupZ-00423
  baseGroupZ must be less than VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]

- VUID-vkCmdDispatchBase-groupCountX-00424
  groupCountX must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0] minus baseGroupX
• VUID-vkCmdDispatchBase-groupCountY-00425
groupCountY must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1] minus baseGroupY

• VUID-vkCmdDispatchBase-groupCountZ-00426
groupCountZ must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2] minus baseGroupZ

• VUID-vkCmdDispatchBase-baseGroupX-00427
If any of baseGroupX, baseGroupY, or baseGroupZ are not zero, then the bound compute pipeline must have been created with the VK_PIPELINE_CREATE_DISPATCH_BASE flag

Valid Usage (Implicit)

• VUID-vkCmdDispatchBase-commandBuffer-parameter
commandBuffer must be a valid VkCommandBuffer handle

• VUID-vkCmdDispatchBase-commandBuffer-recording
commandBuffer must be in the recording state

• VUID-vkCmdDispatchBase-commandBuffer-cmdpool
The VkCommandPool that commandBuffer was allocated from must support compute operations

• VUID-vkCmdDispatchBase-renderpass
This command must only be called outside of a render pass instance

Host Synchronization

• Host access to commandBuffer must be externally synchronized

• Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 31. Device-Generated Commands

This chapter discusses the generation of command buffer content on the device, for which these principle steps are to be taken:

- Define via `VkIndirectCommandsLayoutNV` the sequence of commands which should be generated.
- Optionally make use of device-bindable Shader Groups.
- Retrieve device addresses by `vkGetBufferDeviceAddressEXT` for setting buffers on the device.
- Fill one or more `VkBuffer` with the appropriate content that gets interpreted by `VkIndirectCommandsLayoutNV`.
- Create a preprocess `VkBuffer` using the allocation information from `vkGetGeneratedCommandsMemoryRequirementsNV`.
- Optionally preprocess the input data using `vkCmdPreprocessGeneratedCommandsNV` in a separate action.
- Generate and execute the actual commands via `vkCmdExecuteGeneratedCommandsNV` passing all required data.

`vkCmdPreprocessGeneratedCommandsNV` executes in a separate logical pipeline from either graphics or compute. When preprocessing commands in a separate step they must be explicitly synchronized against the command execution. When not preprocessing, the preprocessing is automatically synchronized against the command execution.

### 31.1. Indirect Commands Layout

The device-side command generation happens through an iterative processing of an atomic sequence comprised of command tokens, which are represented by:

```c
// Provided by VK_NV_device_generated_commands
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkIndirectCommandsLayoutNV)
```

### 31.1.1. Creation and Deletion

Indirect command layouts are created by:

```c
// Provided by VK_NV_device_generated_commands
VkResult vkCreateIndirectCommandsLayoutNV(
    VkDevice                                    device,
    const VkIndirectCommandsLayoutCreateInfoNV* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkIndirectCommandsLayoutNV*                 pIndirectCommandsLayout);
```

- `device` is the logical device that creates the indirect command layout.
- `pCreateInfo` is a pointer to an instance of the `VkIndirectCommandsLayoutCreateInfoNV` structure
containing parameters affecting creation of the indirect command layout.

- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pIndirectCommandsLayout** points to a VkIndirectCommandsLayoutNV handle in which the resulting indirect command layout is returned.

### Valid Usage

- VUID-vkCreateIndirectCommandsLayoutNV-deviceGeneratedCommands-02929
  The VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands feature must be enabled

### Valid Usage (Implicit)

- VUID-vkCreateIndirectCommandsLayoutNV-device-parameter
device must be a valid VkDevice handle
- VUID-vkCreateIndirectCommandsLayoutNV-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkIndirectCommandsLayoutCreateInfoNV structure
- VUID-vkCreateIndirectCommandsLayoutNV-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
- VUID-vkCreateIndirectCommandsLayoutNV-pIndirectCommandsLayout-parameter
  pIndirectCommandsLayout must be a valid pointer to a VkIndirectCommandsLayoutNV handle

### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkIndirectCommandsLayoutCreateInfoNV structure is defined as:
**typedef struct** VkIndirectCommandsLayoutCreateInfoNV {
    VkStructureType                           sType;
    const void*                               pNext;
    VkIndirectCommandsLayoutUsageFlagsNV      flags;
    VkPipelineBindPoint                       pipelineBindPoint;
    uint32_t                                  tokenCount;
    const VkIndirectCommandsLayoutTokenNV*    pTokens;
    uint32_t                                  streamCount;
    const uint32_t*                           pStreamStrides;
} VkIndirectCommandsLayoutCreateInfoNV;

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **pipelineBindPoint** is the **VkPipelineBindPoint** that this layout targets.

• **flags** is a bitmask of **VkIndirectCommandsLayoutUsageFlagBitsNV** specifying usage hints of this layout.

• **tokenCount** is the length of the individual command sequence.

• **pTokens** is an array describing each command token in detail. See **VkIndirectCommandsTokenTypeNV** and **VkIndirectCommandsLayoutTokenNV** below for details.

• **streamCount** is the number of streams used to provide the token inputs.

• **pStreamStrides** is an array defining the byte stride for each input stream.

The following code illustrates some of the flags:
void cmdProcessAllSequences(cmd, pipeline, indirectCommandsLayout, pIndirectCommandsTokens, sequencesCount, indexbuffer, indexbufferOffset)
{
    for (s = 0; s < sequencesCount; s++)
    {
        sUsed = s;

        if (indirectCommandsLayout.flags &
            VK_IN间接DICT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NV) {
            sUsed = indexbuffer.load_uint32( sUsed * sizeof(uint32_t) + indexbufferOffset);
        }

        if (indirectCommandsLayout.flags &
            VK INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NV) {
            sUsed = incoherent_implementation_dependent_permutation[ sUsed ];
        }

        cmdProcessSequence( cmd, pipeline, indirectCommandsLayout, pIndirectCommandsTokens, sUsed );
    }
}
Valid Usage

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-pipelineBindPoint-02930**
  The pipelineBindPoint must be `VK_PIPELINE_BIND_POINT_GRAPHICS`.

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-tokenCount-02931**
  The `tokenCount` must be greater than 0 and less than or equal to `VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxIndirectCommandsTokenCount`.

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-pTokens-02932**
  If `pTokens` contains an entry of `VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV` it must be the first element of the array and there must be only a single element of such token type.

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-pTokens-02933**
  If `pTokens` contains an entry of `VK_INDIRECT_COMMANDS_TOKEN_TYPE_STATE_FLAGS_NV` there must be only a single element of such token type.

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-pTokens-02934**
  All state tokens in `pTokens` must occur prior work provoking tokens (`VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NV`, `VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NV`, `VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_TASKS_NV`).

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-pTokens-02935**
  The content of `pTokens` must include one single work provoking token that is compatible with the pipelineBindPoint.

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-streamCount-02936**
  The `streamCount` must be greater than 0 and less than or equal to `VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxIndirectCommandsStreamCount`.

- **VUID-VkIndirectCommandsLayoutCreateInfoNV-pStreamStrides-02937**
  Each element of `pStreamStrides` must be greater than 0 and less than or equal to `VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxIndirectCommandsStreamStride`. Furthermore the alignment of each token input must be ensured.
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_CREATE_INFO_NV`
- `pNext` must be `NULL`
- `flags` must be a valid combination of `VkIndirectCommandsLayoutUsageFlagBitsNV` values
- `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value
- `pTokens` must be a valid pointer to an array of `tokenCount` valid `VkIndirectCommandsLayoutTokenNV` structures
- `pStreamStrides` must be a valid pointer to an array of `streamCount` `uint32_t` values
- `tokenCount` must be greater than 0
- `streamCount` must be greater than 0

Bits which can be set in `VkIndirectCommandsLayoutCreateInfoNV::flags`, specifying usage hints of an indirect command layout, are:

```c
// Provided by VK_NV_device_generated_commands
typedef enum VkIndirectCommandsLayoutUsageFlagBitsNV {
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EXPLICIT_PREPROCESS_BIT_NV = 0x00000001,
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NV = 0x00000002,
    VK_INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NV = 0x00000004,
} VkIndirectCommandsLayoutUsageFlagBitsNV;
```

- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EXPLICIT_PREPROCESS_BIT_NV` specifies that the layout is always used with the manual preprocessing step through calling `vkCmdPreprocessGeneratedCommandsNV` and executed by `vkCmdExecuteGeneratedCommandsNV` with `isPreprocessed` set to `VK_TRUE`.
- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NV` specifies that the input data for the sequences is not implicitly indexed from 0..`sequencesUsed` but a user provided `VkBuffer` encoding the index is provided.
- `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_UNORDERED_SEQUENCES_BIT_NV` specifies that the processing of sequences can happen at an implementation-dependent order, which is not: guaranteed to be coherent using the same input data.
`typedef VkFlags VkIndirectCommandsLayoutUsageFlagsNV;`

`VkIndirectCommandsLayoutUsageFlagsNV` is a bitmask type for setting a mask of zero or more `VkIndirectCommandsLayoutUsageFlagBitsNV`.

Indirect command layouts are destroyed by:

```c
void vkDestroyIndirectCommandsLayoutNV(
    VkDevice device,
    VkIndirectCommandsLayoutNV indirectCommandsLayout,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the logical device that destroys the layout.
- `indirectCommandsLayout` is the layout to destroy.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- VUID-vkDestroyIndirectCommandsLayoutNV-indirectCommandsLayout-02938
  
  All submitted commands that refer to `indirectCommandsLayout` must have completed execution

- VUID-vkDestroyIndirectCommandsLayoutNV-indirectCommandsLayout-02939
  
  If `VkAllocationCallbacks` were provided when `indirectCommandsLayout` was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyIndirectCommandsLayoutNV-indirectCommandsLayout-02940
  
  If no `VkAllocationCallbacks` were provided when `indirectCommandsLayout` was created, `pAllocator` must be `NULL`

- VUID-vkDestroyIndirectCommandsLayoutNV-deviceGeneratedCommands-02941
  
  The `VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands` feature must be enabled
Valid Usage (Implicit)

- **VUID-vkDestroyIndirectCommandsLayoutNV-device-parameter**
  `device` must be a valid `VkDevice` handle

- **VUID-vkDestroyIndirectCommandsLayoutNV-indirectCommandsLayout-parameter**
  If `indirectCommandsLayout` is not `VK_NULL_HANDLE`, `indirectCommandsLayout` must be a valid `VkIndirectCommandsLayoutNV` handle

- **VUID-vkDestroyIndirectCommandsLayoutNV-pAllocator-parameter**
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkDestroyIndirectCommandsLayoutNV-indirectCommandsLayout-parent**
  If `indirectCommandsLayout` is a valid handle, it must have been created, allocated, or retrieved from `device`

Host Synchronization

- Host access to `indirectCommandsLayout` must be externally synchronized

31.1.2. Token Input Streams

The `VkIndirectCommandsStreamNV` structure specifies the input data for one or more tokens at processing time.

```c
// Provided by VK_NV_device_generated_commands
typedef struct VkIndirectCommandsStreamNV {
  VkBuffer      buffer;
  VkDeviceSize  offset;
} VkIndirectCommandsStreamNV;
```

- `buffer` specifies the `VkBuffer` storing the functional arguments for each sequence. These arguments can be written by the device.
- `offset` specified an offset into `buffer` where the arguments start.
Valid Usage

• VUID-VkIndirectCommandsStreamNV-buffer-02942
  The buffer's usage flag must have the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

• VUID-VkIndirectCommandsStreamNV-offset-02943
  The offset must be aligned to VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::minIndirectCommandsBufferOffsetAlignment

• VUID-VkIndirectCommandsStreamNV-buffer-02975
  If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

Valid Usage (Implicit)

• VUID-VkIndirectCommandsStreamNV-buffer-parameter
  buffer must be a valid VkBuffer handle

The input streams can contain raw uint32_t values, existing indirect commands such as:

• VkDrawIndirectCommand
• VkDrawIndexedIndirectCommand
• VkDrawMeshTasksIndirectCommandNV

or additional commands as listed below. How the data is used is described in the next section.

The VkBindShaderGroupIndirectCommandNV structure specifies the input data for the VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV token.

```
// Provided by VK_NV_device_generated_commands
typedef struct VkBindShaderGroupIndirectCommandNV {
  uint32_t groupIndex;
} VkBindShaderGroupIndirectCommandNV;
```

• index specifies which shader group of the current bound graphics pipeline is used.

Valid Usage

• VUID-VkBindShaderGroupIndirectCommandNV-None-02944
  The current bound graphics pipeline, as well as the pipelines it may reference, must have been created with VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV

• VUID-VkBindShaderGroupIndirectCommandNV-index-02945
  The index must be within range of the accessible shader groups of the current bound graphics pipeline. See vkCmdBindPipelineShaderGroupNV for further details
The `VkBindIndexBufferIndirectCommandNV` structure specifies the input data for the `VK_INDIRECT_COMMANDS_TOKEN_TYPE_INDEX_BUFFER_NV` token.

```c
// Provided by VK_NV_device_generated_commands
typedef struct VkBindIndexBufferIndirectCommandNV {
    VkDeviceAddress    bufferAddress;
    uint32_t           size;
    VkIndexType        indexType;
} VkBindIndexBufferIndirectCommandNV;
```

- `bufferAddress` specifies a physical address of the `VkBuffer` used as index buffer.
- `size` is the byte size range which is available for this operation from the provided address.
- `indexType` is a `VkIndexType` value specifying how indices are treated. Instead of the Vulkan enum values, a custom `uint32_t` value can be mapped to an `VkIndexType` by specifying the `VkIndirectCommandsLayoutTokenNV::pIndexTypes` and `VkIndirectCommandsLayoutTokenNV::pIndexTypeValues` arrays.

**Valid Usage**

- **VUID-VkBindIndexBufferIndirectCommandNV-None-02946**
  The buffer's usage flag from which the address was acquired must have the `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` bit set

- **VUID-VkBindIndexBufferIndirectCommandNV-bufferAddress-02947**
  The `bufferAddress` must be aligned to the `indexType` used

- **VUID-VkBindIndexBufferIndirectCommandNV-None-02948**
  Each element of the buffer from which the address was acquired and that is non-sparse must be bound completely and contiguously to a single `VkDeviceMemory` object

**Valid Usage (Implicit)**

- **VUID-VkBindIndexBufferIndirectCommandNV-indexType-parameter**
  `indexType` must be a valid `VkIndexType` value

The `VkBindVertexBufferIndirectCommandNV` structure specifies the input data for the `VK_INDIRECT_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NV` token.

```c
// Provided by VK_NV_device_generated_commands
typedef struct VkBindVertexBufferIndirectCommandNV {
    VkDeviceAddress    bufferAddress;
    uint32_t           size;
    uint32_t           stride;
} VkBindVertexBufferIndirectCommandNV;
```
• `bufferAddress` specifies a physical address of the `VkBuffer` used as vertex input binding.
• `size` is the byte size range which is available for this operation from the provided address.
• `stride` is the byte size stride for this vertex input binding as in `VkVertexInputBindingDescription::stride`. It is only used if `VkIndirectCommandsLayoutTokenNV::vertexDynamicStride` was set, otherwise the stride is inherited from the current bound graphics pipeline.

### Valid Usage

- **VUID-VkBindVertexBufferIndirectCommandNV-None-02949**
  The buffer’s usage flag from which the address was acquired must have the `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` bit set
- **VUID-VkBindVertexBufferIndirectCommandNV-None-02950**
  Each element of the buffer from which the address was acquired and that is non-sparse must be bound completely and contiguously to a single `VkDeviceMemory` object

The `VkSetStateFlagsIndirectCommandNV` structure specifies the input data for the `VK_INDIRECT_COMMANDS_TOKEN_TYPE_STATE_FLAGS_NV` token. Which state is changed depends on the `VkIndirectStateFlagBitsNV` specified at `VkIndirectCommandsLayoutNV` creation time.

```c
// Provided by VK_NV_device_generated_commands
typedef struct VkSetStateFlagsIndirectCommandNV {
    uint32_t data;
} VkSetStateFlagsIndirectCommandNV;
```

- `data` encodes packed state that this command alters.
  - Bit 0: If set represents `VK_FRONT_FACE_CLOCKWISE`, otherwise `VK_FRONT_FACE_COUNTER_CLOCKWISE`

A subset of the graphics pipeline state can be altered using indirect state flags:

```c
// Provided by VK_NV_device_generated_commands
typedef enum VkIndirectStateFlagBitsNV {
    VK_INDIRECT_STATE_FLAG_FRONTFACE_BIT_NV = 0x00000001,
} VkIndirectStateFlagBitsNV;
```

- `VK_INDIRECT_STATE_FLAG_FRONTFACE_BIT_NV` allows to toggle the `VkFrontFace` rasterization state for subsequent draw operations.

```c
// Provided by VK_NV_device_generated_commands
typedef VkFlags VkIndirectStateFlagsNV;
```

`VkIndirectStateFlagsNV` is a bitmask type for setting a mask of zero or more `VkIndirectStateFlagBitsNV`. 
31.1.3. Tokenized Command Processing

The processing is in principle illustrated below:

```c
void cmdProcessSequence(cmd, pipeline, indirectCommandsLayout,
                        pIndirectCommandsStreams, s)
{
    for (t = 0; t < indirectCommandsLayout.tokenCount; t++)
    {
        uint32_t stream   = indirectCommandsLayout.pTokens[t].stream;
        uint32_t offset   = indirectCommandsLayout.pTokens[t].offset;
        uint32_t stride   = indirectCommandsLayout.pStreamStrides[stream];
        stream            = pIndirectCommandsStreams[stream];
        const void* input = stream.buffer.pointer( stream.offset + stride * s + offset )

        // further details later
        indirectCommandsLayout.pTokens[t].command (cmd, pipeline, input, s);
    }
}

void cmdProcessAllSequences(cmd, pipeline, indirectCommandsLayout,
                             pIndirectCommandsStreams, sequencesCount)
{
    for (s = 0; s < sequencesCount; s++)
    {
        cmdProcessSequence(cmd, pipeline, indirectCommandsLayout,
                            pIndirectCommandsStreams, s);
    }
}
```

The processing of each sequence is considered stateless, therefore all state changes must occur prior work provoking commands within the sequence. A single sequence is strictly targeting the VkPipelineBindPoint it was created with.

The primary input data for each token is provided through VkBuffer content at preprocessing using `vkCmdPreprocessGeneratedCommandsNV` or execution time using `vkCmdExecuteGeneratedCommandsNV`, however some functional arguments, for example binding sets, are specified at layout creation time. The input size is different for each token.

Possible values of those elements of the `VkIndirectCommandsLayoutCreateInfoNV::pTokens` array which specify command tokens (other elements of the array specify command parameters) are:
typedef enum VkIndirectCommandsTokenTypeNV {
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV = 0,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_STATE_FLAGS_NV = 1,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_INDEX_BUFFER_NV = 2,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NV = 3,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV = 4,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NV = 5,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NV = 6,
    VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_TASKS_NV = 7,
} VkIndirectCommandsTokenTypeNV;

**Table 45. Supported indirect command tokens**

<table>
<thead>
<tr>
<th>Token type</th>
<th>Equivalent command</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV</td>
<td>vkCmdBindPipelineShaderGroupNV</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_STATE_FLAGS_NV</td>
<td>-</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_INDEX_BUFFER_NV</td>
<td>vkCmdBindIndexBuffer</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NV</td>
<td>vkCmdBindVertexBuffer</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV</td>
<td>vkCmdPushConstants</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NV</td>
<td>vkCmdDrawIndexedIndirect</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NV</td>
<td>vkCmdDrawIndirect</td>
</tr>
<tr>
<td>VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_TASKS_NV</td>
<td>vkCmdDrawMeshTasksIndirectNV</td>
</tr>
</tbody>
</table>

The VkIndirectCommandsLayoutTokenNV structure specifies details to the function arguments that need to be known at layout creation time:
typedef struct VkIndirectCommandsLayoutTokenNV {
    VkStructureType sType;
    const void* pNext;
    VkIndirectCommandsTokenTypeNV tokenType;
    uint32_t stream;
    uint32_t offset;
    uint32_t vertexBindingUnit;
    VkBool32 vertexDynamicStride;
    VkPipelineLayout pushconstantPipelineLayout;
    VkShaderStageFlags pushconstantShaderStageFlags;
    uint32_t pushconstantOffset;
    uint32_t pushconstantSize;
    VkIndirectStateFlagsNV indirectStateFlags;
    uint32_t indexTypeCount;
    const VkIndexType* pIndexTypes;
    const uint32_t* pIndexTypeValues;
} VkIndirectCommandsLayoutTokenNV;

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `tokenType` specifies the token command type.
- `stream` is the index of the input stream that contains the token argument data.
- `offset` is a relative starting offset within the input stream memory for the token argument data.
- `vertexBindingUnit` is used for the vertex buffer binding command.
- `vertexDynamicStride` sets if the vertex buffer stride is provided by the binding command rather than the current bound graphics pipeline state.
- `pushconstantPipelineLayout` is the `VkPipelineLayout` used for the push constant command.
- `pushconstantShaderStageFlags` are the shader stage flags used for the push constant command.
- `pushconstantOffset` is the offset used for the push constant command.
- `pushconstantSize` is the size used for the push constant command.
- `indirectStateFlags` are the active states for the state flag command.
- `indexTypeCount` is the optional size of the `pIndexTypes` and `pIndexTypeValues` array pairings. If not zero, it allows to register a custom `uint32_t` value to be treated as specific `VkIndexType`.
- `pIndexTypes` is the used `VkIndexType` for the corresponding `uint32_t` value entry in `pIndexTypeValues`. 
Valid Usage

- VUID-VkIndirectCommandsLayoutTokenNV-stream-02951
  stream must be smaller than VkIndirectCommandsLayoutCreateInfoNV::streamCount

- VUID-VkIndirectCommandsLayoutTokenNV-offset-02952
  offset must be less than or equal to
  VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxIndirectCommandsTokenOffset

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02976
  If tokenType is VK_INDEED_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NV, vertexBindingUnit must stay within device supported limits for the appropriate commands

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02977
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, pushconstantPipelineLayout must be valid

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02978
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, pushconstantOffset must be a multiple of 4

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02979
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, pushconstantSize must be a multiple of 4

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02980
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, pushconstantOffset must be less than
  VkPhysicalDeviceLimits::maxPushConstantsSize

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02981
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, pushconstantSize must be less than or equal to
  VkPhysicalDeviceLimits::maxPushConstantsSize minus pushconstantOffset

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02982
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, for each byte in the range specified by
  pushconstantOffset and pushconstantSize and for each shader stage in pushconstantShaderStageFlags, there must be a push constant range in
  pushconstantPipelineLayout that includes that byte and that stage

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02983
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, for each byte in the range specified by pushconstantOffset and
  pushconstantSize and for each push constant range that overlaps that byte, pushconstantShaderStageFlags must include all stages in that
  push constant range'sVkPushConstantRange::pushconstantShaderStageFlags

- VUID-VkIndirectCommandsLayoutTokenNV-tokenType-02984
  If tokenType is VK_INDIRECT_COMMANDS_TOKEN_TYPE_STATE_FLAGS_NV, indirectStateFlags must not be 0
Valid Usage (Implicit)

- **VUID-VkIndirectCommandsLayoutTokenNV-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_TOKEN_NV`

- **VUID-VkIndirectCommandsLayoutTokenNV-pNext-pNext**
  
  `pNext` **must** be `NULL`

- **VUID-VkIndirectCommandsLayoutTokenNV-tokenType-parameter**
  
  `tokenType` **must** be a valid `VkIndirectCommandsTokenTypeNV` value

- **VUID-VkIndirectCommandsLayoutTokenNV-pushconstantPipelineLayout-parameter**
  
  If `pushconstantPipelineLayout` is not `VK_NULL_HANDLE`, `pushconstantPipelineLayout` **must** be a valid `VkPipelineLayout` handle

- **VUID-VkIndirectCommandsLayoutTokenNV-pushconstantShaderStageFlags-parameter**
  
  `pushconstantShaderStageFlags` **must** be a valid combination of `VkShaderStageFlagBits` values

- **VUID-VkIndirectCommandsLayoutTokenNV-indirectStateFlags-parameter**
  
  `indirectStateFlags` **must** be a valid combination of `VkIndirectStateFlagBitsNV` values

- **VUID-VkIndirectCommandsLayoutTokenNV-pIndexTypes-parameter**
  
  If `indexTypeCount` is not `0`, `pIndexTypes` **must** be a valid pointer to an array of `VkIndexType` values

- **VUID-VkIndirectCommandsLayoutTokenNV-pIndexTypeValues-parameter**
  
  If `indexTypeCount` is not `0`, `pIndexTypeValues` **must** be a valid pointer to an array of `indexTypeCount uint32_t` values

The following code provides detailed information on how an individual sequence is processed. For valid usage, all restrictions from the regular commands apply.

```c
void cmdProcessSequence(cmd, pipeline, indirectCommandsLayout, pIndirectCommandsStreams, s)
{
    for (uint32_t t = 0; t < indirectCommandsLayout.tokenCount; t++){
        token = indirectCommandsLayout.pTokens[t];

        uint32_t stride   = indirectCommandsLayout.pStreamStrides[token.stream];
        stream            = pIndirectCommandsStreams[token.stream];
        uint32_t offset   = stream.offset + stride * s + token.offset;
        const void* input = stream.buffer.pointer( offset )

        switch(input.type){
            case VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV:
                VkBindShaderGroupIndirectCommandNV* bind = input;

                vkCmdBindPipelineShaderGroupNV(cmd, indirectCommandsLayout.pipelineBindPoint, pipeline, bind->groupIndex);
            break;
        }
    }
}
```

1648 | Chapter 31. Device-Generated Commands
VK_INDIRECT_COMMANDS_TOKEN_TYPE_STATE_FLAGS_NV:
  VkSetStateFlagsIndirectCommandNV* state = input;

  if (token.indirectStateFlags & VK_INDIRECT_STATE_FLAG_FRONTFACE_BIT_NV){
    if (state.data & (1 << 0)){
      set VK_FRONT_FACE_CLOCKWISE;
    } else {
      set VK_FRONT_FACE_COUNTER_CLOCKWISE;
    }
  }

  break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV:
  uint32_t* data = input;

  vkCmdPushConstants(cmd,
    token.pushconstantPipelineLayout
token.pushconstantStageFlags,
token.pushconstantOffset,
token.pushconstantSize, data);

  break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_INDEX_BUFFER_NV:
  VkBindIndexBufferIndirectCommandNV* data = input;

  // the indexType may optionally be remapped
  // from a custom uint32_t value, via
  // VkIndirectCommandsLayoutTokenNV::pIndexTypeValues

  vkCmdBindIndexBuffer(cmd,
    deriveBuffer(data->bufferAddress),
    deriveOffset(data->bufferAddress),
    data->indexType);

  break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_VERTEX_BUFFER_NV:
  VkBindVertexBufferIndirectCommandNV* data = input;

  // if token.vertexDynamicStride is VK_TRUE
  // then the stride for this binding is set
  // using data->stride as well

  vkCmdBindVertexBuffers(cmd,
    token.vertexBindingUnit, 1,
    &deriveBuffer(data->bufferAddress),
    &deriveOffset(data->bufferAddress));

  break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_INDEXED_NV:
  vkCmdDrawIndexedIndirect(cmd,
    stream.buffer, offset, 1, 0);
break;

VK_INDIRECT_COMMANDS_TOKEN_TYPE_DRAW_NV:
  vkCmdDrawIndirect(cmd,
            stream.buffer,  
            offset, 1, 0);
break;

// only available if VK_NV_mesh_shader is supported
VK_INDIRECT_COMMANDS_TOKEN_TYPE_DISPATCH_NV:
  vkCmdDrawMeshTasksIndirectNV(cmd,
            stream.buffer, offset, 1, 0);
break;
}
}

31.2. Indirect Commands Generation And Execution

The generation of commands on the device requires a preprocess buffer. To retrieve the memory size and alignment requirements of a particular execution state call:

```c
// Provided by VK_NV_device_generated_commands
void vkGetGeneratedCommandsMemoryRequirementsNV(
  VkDevice                                    device,
  const VkGeneratedCommandsMemoryRequirementsInfoNV* pInfo,
  VkMemoryRequirements2*                      pMemoryRequirements);
```

- `device` is the logical device that owns the buffer.
- `pInfo` is a pointer to an instance of the `VkGeneratedCommandsMemoryRequirementsInfoNV` structure containing parameters required for the memory requirements query.
- `pMemoryRequirements` points to an instance of the `VkMemoryRequirements2` structure in which the memory requirements of the buffer object are returned.

**Valid Usage**

- `VUID-vkGetGeneratedCommandsMemoryRequirementsNV-deviceGeneratedCommands-02906`
  The `VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands` feature must be enabled
Valid Usage (Implicit)

- VUID-vkGetGeneratedCommandsMemoryRequirementsNV-device-parameter
  
  `device` must be a valid `VkDevice` handle

- VUID-vkGetGeneratedCommandsMemoryRequirementsNV-pInfo-parameter
  
  `pInfo` must be a valid pointer to a valid `VkGeneratedCommandsMemoryRequirementsInfoNV` structure

- VUID-vkGetGeneratedCommandsMemoryRequirementsNV-pMemoryRequirements-parameter
  
  `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements2` structure

// Provided by VK_NV_device_generated_commands

typedef struct VkGeneratedCommandsMemoryRequirementsInfoNV {
    VkStructureType               sType;
    const void*                   pNext;
    VkPipelineBindPoint           pipelineBindPoint;
    VkPipeline                    pipeline;
    VkIndirectCommandsLayoutNV    indirectCommandsLayout;
    uint32_t                      maxSequencesCount;
} VkGeneratedCommandsMemoryRequirementsInfoNV;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pipelineBindPoint` is the `VkPipelineBindPoint` of the `pipeline` that this buffer memory is intended to be used with during the execution.
- `pipeline` is the `VkPipeline` that this buffer memory is intended to be used with during the execution.
- `indirectCommandsLayout` is the `VkIndirectCommandsLayoutNV` that this buffer memory is intended to be used with.
- `maxSequencesCount` is the maximum number of sequences that this buffer memory in combination with the other state provided can be used with.

Valid Usage

- VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-maxSequencesCount-02907
  
  `maxSequencesCount` must be less or equal to `VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxIndirectSequenceCount`
Valid Usage (Implicit)

• VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_GENERATED_COMMANDS_MEMORY_REQUIREMENTS_INFO_NV

• VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-pNext-pNext
  pNext must be NULL

• VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-pipelineBindPoint-parameter
  pipelineBindPoint must be a valid VkPipelineBindPoint value

• VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-pipeline-parameter
  pipeline must be a valid VkPipeline handle

• VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-indirectCommandsLayout-parameter
  indirectCommandsLayout must be a valid VkIndirectCommandsLayoutNV handle

• VUID-VkGeneratedCommandsMemoryRequirementsInfoNV-commonparent
  Both of indirectCommandsLayout, and pipeline must have been created, allocated, or
  retrieved from the same VkDevice

The actual generation of commands as well as their execution on the device is handled as single
action with:

```c
// Provided by VK_NV_device_generated_commands
void vkCmdExecuteGeneratedCommandsNV(
    VkCommandBuffer                             commandBuffer,
    VkBool32                                    isPreprocessed,
    const VkGeneratedCommandsInfoNV*            pGeneratedCommandsInfo);
```

• `commandBuffer` is the command buffer into which the command is recorded.

• `isPreprocessed` represents whether the input data has already been preprocessed on the device.
  If it is VK_FALSE this command will implicitly trigger the preprocessing step, otherwise not.

• `pGeneratedCommandsInfo` is a pointer to an instance of the VkGeneratedCommandsInfoNV
  structure containing parameters affecting the generation of commands.
Valid Usage

- **VUID-vkCmdExecuteGeneratedCommandsNV-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdExecuteGeneratedCommandsNV-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdExecuteGeneratedCommandsNV-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdExecuteGeneratedCommandsNV-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view’s format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- **VUID-vkCmdExecuteGeneratedCommandsNV-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdExecuteGeneratedCommandsNV-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdExecuteGeneratedCommandsNV-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- **VUID-vkCmdExecuteGeneratedCommandsNV-None-02697**
  For each set $n$ that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to $n$ at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set $n$, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- **VUID-vkCmdExecuteGeneratedCommandsNV-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility
Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

A valid pipeline must be bound to the pipeline bind point used by this command.

If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

If `commandBuffer` is an unprotected command buffer, any resource accessed by the
VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

• VUID-vkCmdExecuteGeneratedCommandsNV-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

• VUID-vkCmdExecuteGeneratedCommandsNV-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

• VUID-vkCmdExecuteGeneratedCommandsNV-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

• VUID-vkCmdExecuteGeneratedCommandsNV-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

• VUID-vkCmdExecuteGeneratedCommandsNV-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

• VUID-vkCmdExecuteGeneratedCommandsNV-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

• VUID-vkCmdExecuteGeneratedCommandsNV-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

• VUID-vkCmdExecuteGeneratedCommandsNV-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

• VUID-vkCmdExecuteGeneratedCommandsNV-renderPass-02684
  The current render pass must be compatible with the renderPass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-vkCmdExecuteGeneratedCommandsNV-subpass-02685
  The subpass index of the current render pass must be equal to the subpass member of the VkGraphicsPipelineCreateInfo structure specified when creating the VkPipeline bound to VK_PIPELINE_BIND_POINT_GRAPHICS

• VUID-vkCmdExecuteGeneratedCommandsNV-None-02686
  Every input attachment used by the current subpass must be bound to the pipeline via a
descriptor set

- **VUID-vkCmdExecuteGeneratedCommandsNV-None-04584**
  Image subresources used as attachments in the current render pass must not be accessed in any way other than as an attachment by this command, except for cases involving read-only access to depth/stencil attachments as described in the Render Pass chapter

- **VUID-vkCmdExecuteGeneratedCommandsNV-maxMultiviewInstanceIndex-02688**
  If the draw is recorded in a render pass instance with multiview enabled, the maximum instance index must be less than or equal to VkPhysicalDeviceMultiviewProperties::maxMultiviewInstanceIndex

- **VUID-vkCmdExecuteGeneratedCommandsNV-sampleLocationsEnable-02689**
  If the bound graphics pipeline was created with VkPipelineSampleLocationsStateCreateInfoEXT::sampleLocationsEnable set to VK_TRUE and the current subpass has a depth/stencil attachment, then that attachment must have been created with the VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT bit set

- **VUID-vkCmdExecuteGeneratedCommandsNV-viewportCount-03417**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetViewportWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the VkPipelineViewportStateCreateInfo::viewportCount of the pipeline

- **VUID-vkCmdExecuteGeneratedCommandsNV-scissorCount-03418**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, then vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the scissorCount parameter of vkCmdSetScissorWithCountEXT must match the viewportCount parameter of vkCmdSetViewportWithCountEXT

- **VUID-vkCmdExecuteGeneratedCommandsNV-viewportCount-03419**
  If the bound graphics pipeline state was created with both the VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT and VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic states enabled then both vkCmdSetViewportWithCountEXT and vkCmdSetScissorWithCountEXT must have been called in the current command buffer prior to this draw command, and the viewportCount parameter of vkCmdSetViewportWithCountEXT must match the scissorCount parameter of vkCmdSetScissorWithCountEXT

- **VUID-vkCmdExecuteGeneratedCommandsNV-viewportCount-04137**
  If the bound graphics pipeline state was created with the VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT dynamic state enabled, but not the VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV dynamic state enabled, then the bound graphics pipeline must have been created with VkPipelineViewportWScalingStateCreateInfoNV::viewportCount greater or equal to the viewportCount parameter in the last call to vkCmdSetViewportWithCountEXT
If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportWScalingNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled, but not the `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic state enabled, then the bound graphics pipeline must have been created with `VkPipelineViewportShadingRateImageStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` and `VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV` dynamic states enabled then the `viewportCount` parameter in the last call to `vkCmdSetViewportShadingRatePaletteNV` must be greater than or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportSwizzleStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportSwizzleStateCreateInfoNV::viewportCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT` dynamic state enabled and an instance of `VkPipelineViewportExclusiveScissorStateCreateInfoNV` chained from `VkPipelineViewportCreateInfo`, then the bound graphics pipeline must have been created with `VkPipelineViewportExclusiveScissorStateCreateInfoNV::exclusiveScissorCount` greater or equal to the `viewportCount` parameter in the last call to `vkCmdSetViewportWithCountEXT`.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT` dynamic state enabled then `vkCmdSetPrimitiveTopologyEXT` must have been called in the current command buffer prior to this draw command, and the `primitiveTopology` parameter of `vkCmdSetPrimitiveTopologyEXT` must be of the same topology class as the pipeline `VkPipelineInputAssemblyStateCreateInfo::topology` state.

If the bound graphics pipeline state was created with the `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT` dynamic state enabled then...
\textbf{vkCmdSetPatchControlPointsEXT must} have been called in the current command buffer prior to this draw command

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-None-04876
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT} dynamic state enabled then \texttt{vkCmdSetRasterizerDiscardEnableEXT must} have been called in the current command buffer prior to this draw command

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-None-04877
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT} dynamic state enabled then \texttt{vkCmdSetDepthBiasEnableEXT must} have been called in the current command buffer prior to this draw command

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-logicOp-04878
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_LOGIC_OP_EXT} dynamic state enabled then \texttt{vkCmdSetLogicOpEXT must} have been called in the current command buffer prior to this draw command and the \texttt{logicOp must} be a valid \texttt{VkLogicOp} value

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-None-04879
  If the bound graphics pipeline state was created with the \texttt{VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT} dynamic state enabled then \texttt{vkCmdSetPrimitiveRestartEnableEXT must} have been called in the current command buffer prior to this draw command

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-primitiveFragmentShadingRateWithMultipleViewports-04552
  If the \texttt{primitiveFragmentShadingRateWithMultipleViewports} limit is not supported, the bound graphics pipeline was created with the \texttt{VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT} dynamic state enabled, and any of the shader stages of the bound graphics pipeline write to the \texttt{PrimitiveShadingRateKHR} built-in, then \texttt{vkCmdSetViewportWithCountEXT must} have been called in the current command buffer prior to this draw command, and the \texttt{viewportCount} parameter of \texttt{vkCmdSetViewportWithCountEXT must} be \texttt{1}

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-blendEnable-04727
  If rasterization is not disabled in the bound graphics pipeline, then for each color attachment in the subpass, if the corresponding image view’s \texttt{format features} do not contain \texttt{VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT}, then the \texttt{blendEnable} member of the corresponding element of the \texttt{pAttachments} member of \texttt{pColorBlendState must} be \texttt{VK_FALSE}

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-rasterizationSamples-04740
  If rasterization is not disabled in the bound graphics pipeline, and neither the \texttt{VK_AMD_mixed_attachment_samples} nor the \texttt{VK_NV_framebuffer_mixed_samples} extensions are enabled, then \texttt{VkPipelineMultisampleStateCreateInfo::rasterizationSamples must} be the same as the current subpass color and/or depth/stencil attachments

- \textbullet\ VUID-vkCmdExecuteGeneratedCommandsNV-pStrides-04884
  If the bound graphics pipeline was created with the \texttt{VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT} dynamic state enabled, then \texttt{vkCmdBindVertexBuffers2EXT must} have been called in the current command buffer prior to this draw command, and the \texttt{pStrides} parameter of
\textbf{vkCmdBindVertexBuffers2EXT} must \textbf{not be} \textbf{NULL}

- VUID-vkCmdExecuteGeneratedCommandsNV-None-04007
  All vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface \textbf{must} have either valid or \textbf{VK_NULL_HANDLE} buffers bound

- VUID-vkCmdExecuteGeneratedCommandsNV-None-04008
  If the \textbf{nullDescriptor} feature is not enabled, all vertex input bindings accessed via vertex input variables declared in the vertex shader entry point's interface \textbf{must} \textbf{not be} \textbf{VK_NULL_HANDLE}

- VUID-vkCmdExecuteGeneratedCommandsNV-None-02721
  For a given vertex buffer binding, any attribute data fetched \textbf{must} be entirely contained within the corresponding vertex buffer binding, as described in \textbf{Vertex Input Description}

- VUID-vkCmdExecuteGeneratedCommandsNV-commandBuffer-02970
  \textbf{commandBuffer} \textbf{must} \textbf{not be} a protected command buffer

- VUID-vkCmdExecuteGeneratedCommandsNV-isPreprocessed-02908
  If \textbf{isPreprocessed} is \textbf{VK_TRUE} then \textbf{vkCmdPreprocessGeneratedCommandsNV} \textbf{must} have already been executed on the device, using the same \textbf{pGeneratedCommandsInfo} content as well as the content of the input buffers it references (all except \textbf{VkGeneratedCommandsInfoNV::preprocessBuffer}). Furthermore \textbf{pGeneratedCommandsInfo}’s \textbf{indirectCommandsLayout} \textbf{must} \textbf{have been} \textbf{created with} the \textbf{VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EXPLICIT_PREPROCESS_BIT_NV} bit set

- VUID-vkCmdExecuteGeneratedCommandsNV-pipeline-02909
  \textbf{VkGeneratedCommandsInfoNV::pipeline} \textbf{must} match the current bound pipeline at \textbf{VkGeneratedCommandsInfoNV::pipelineBindPoint}

- VUID-vkCmdExecuteGeneratedCommandsNV-None-02910
  \textbf{Transform feedback} \textbf{must} \textbf{not be} \textbf{active}

- VUID-vkCmdExecuteGeneratedCommandsNV-deviceGeneratedCommands-02911
  The \textbf{VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands} feature \textbf{must} \textbf{be} \textbf{enabled}
Valid Usage (Implicit)

- VUID-vkCmdExecuteGeneratedCommandsNV-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdExecuteGeneratedCommandsNV-pGeneratedCommandsInfo-parameter
  pGeneratedCommandsInfo must be a valid pointer to a valid VkGeneratedCommandsInfoNV structure

- VUID-vkCmdExecuteGeneratedCommandsNV-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdExecuteGeneratedCommandsNV-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

- VUID-vkCmdExecuteGeneratedCommandsNV-renderpass
  This command must only be called inside of a render pass instance

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Inside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>
typedef struct VkGeneratedCommandsInfoNV {
    VkStructureType                      sType;
    const void*                          pNext;
    VkPipelineBindPoint                  pipelineBindPoint;
    VkPipeline                           pipeline;
    VkIndirectCommandsLayoutNV           indirectCommandsLayout;
    uint32_t                             streamCount;
    const VkIndirectCommandsStreamNV*    pStreams;
    uint32_t                             sequencesCount;
    VkBuffer                             preprocessBuffer;
    VkDeviceSize                         preprocessOffset;
    VkDeviceSize                         preprocessSize;
    VkBuffer                             sequencesCountBuffer;
    VkDeviceSize                         sequencesCountOffset;
    VkBuffer                             sequencesIndexBuffer;
    VkDeviceSize                         sequencesIndexOffset;
} VkGeneratedCommandsInfoNV;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pipelineBindPoint** is the VkPipelineBindPoint used for the pipeline.
- **pipeline** is the VkPipeline used in the generation and execution process.
- **indirectCommandsLayout** is the VkIndirectCommandsLayoutNV that provides the command sequence to generate.
- **streamCount** defines the number of input streams
- **pStreams** provides an array of VkIndirectCommandsStreamNV that provide the input data for the tokens used in indirectCommandsLayout.
- **sequencesCount** is the maximum number of sequences to reserve. If sequencesCountBuffer is VK_NULL_HANDLE, this is also the actual number of sequences generated.
- **preprocessBuffer** is the VkBuffer that is used for preprocessing the input data for execution. If this structure is used with vkCmdExecuteGeneratedCommandsNV with its isPreprocessed set to VK_TRUE, then the preprocessing step is skipped and data is only read from this buffer.
- **preprocessOffset** is the byte offset into preprocessBuffer where the preprocessed data is stored.
- **preprocessSize** is the maximum byte size within the preprocessBuffer after the preprocessOffset that is available for preprocessing.
- **sequencesCountBuffer** is a VkBuffer in which the actual number of sequences is provided as single uint32_t value.
- **sequencesCountOffset** is the byte offset into sequencesCountBuffer where the count value is stored.
- **sequencesIndexBuffer** is a VkBuffer that encodes the used sequence indices as uint32_t array.
- **sequencesIndexOffset** is the byte offset into sequencesIndexBuffer where the index values start.
Valid Usage

- VUID-VkGeneratedCommandsInfoNV-pipeline-02912
  The provided pipeline must match the pipeline bound at execution time.

- VUID-VkGeneratedCommandsInfoNV-indirectCommandsLayout-02913
  If the indirectCommandsLayout uses a token of
  VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV, then the pipeline must have been
  created with multiple shader groups.

- VUID-VkGeneratedCommandsInfoNV-indirectCommandsLayout-02914
  If the indirectCommandsLayout uses a token of
  VK_INDIRECT_COMMANDS_TOKEN_TYPE_SHADER_GROUP_NV, then the pipeline must have been
  created with VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV set in
  VkGraphicsPipelineCreateInfo::flags.

- VUID-VkGeneratedCommandsInfoNV-indirectCommandsLayout-02915
  If the indirectCommandsLayout uses a token of
  VK_INDIRECT_COMMANDS_TOKEN_TYPE_PUSH_CONSTANT_NV, then the pipeline's VkPipelineLayout
  must match the VkIndirectCommandsLayoutTokenNV::pushconstantPipelineLayout.

- VUID-VkGeneratedCommandsInfoNV-streamCount-02916
  streamCount must match the indirectCommandsLayout's streamCount.

- VUID-VkGeneratedCommandsInfoNV-sequencesCount-02917
  sequencesCount must be less or equal to
  VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::maxIndirectSequenceCount and
  VkGeneratedCommandsMemoryRequirementsInfoNV::maxSequencesCount that was used to
determine the preprocessSize.

- VUID-VkGeneratedCommandsInfoNV-preprocessBuffer-02918
  preprocessBuffer must have the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set in its usage
  flag.

- VUID-VkGeneratedCommandsInfoNV-preprocessOffset-02919
  preprocessOffset must be aligned to
  VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::minIndirectCommandsBufferOff
  setAlignment.

- VUID-VkGeneratedCommandsInfoNV-preprocessBuffer-02971
  If preprocessBuffer is non-sparse then it must be bound completely and contiguously to a
  single VkDeviceMemory object.

- VUID-VkGeneratedCommandsInfoNV-preprocessSize-02920
  preprocessSize must be at least equal to the memory requirement's size returned by
  vkGetGeneratedCommandsMemoryRequirementsNV using the matching inputs
  (indirectCommandsLayout, ...) as within this structure.

- VUID-VkGeneratedCommandsInfoNV-sequencesCountBuffer-02921
  sequencesCountBuffer can be set if the actual used count of sequences is sourced from the
  provided buffer. In that case the sequencesCount serves as upper bound.

- VUID-VkGeneratedCommandsInfoNV-sequencesCountBuffer-02922
  If sequencesCountBuffer is not VK_NULL_HANDLE, its usage flag must have the
VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-VkGeneratedCommandsInfoNV-sequencesCountBuffer-02923
  If sequencesCountBuffer is not VK_NULL_HANDLE, sequencesCountOffset must be aligned to VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::minSequencesCountBufferOffsetAlignment

- VUID-VkGeneratedCommandsInfoNV-sequencesCountBuffer-02972
  If sequencesCountBuffer is not VK_NULL_HANDLE and is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-VkGeneratedCommandsInfoNV-sequencesIndexBuffer-02924
  If indirectCommandsLayout's VK INDIRECT_COMMANDS_LAYOUT_USAGE_INDEXED_SEQUENCES_BIT_NV is set, sequencesIndexBuffer must be set otherwise it must be VK_NULL_HANDLE

- VUID-VkGeneratedCommandsInfoNV-sequencesIndexBuffer-02925
  If sequencesIndexBuffer is not VK_NULL_HANDLE, its usage flag must have the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set

- VUID-VkGeneratedCommandsInfoNV-sequencesIndexBuffer-02926
  If sequencesIndexBuffer is not VK_NULL_HANDLE, sequencesIndexOffset must be aligned to VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV::minSequencesIndexBufferOffsetAlignment

- VUID-VkGeneratedCommandsInfoNV-sequencesIndexBuffer-02973
  If sequencesIndexBuffer is not VK_NULL_HANDLE and is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object
Valid Usage (Implicit)

- VUID-VkGeneratedCommandsInfoNV-sType-sType
  
  **sType** **must** be **VK_STRUCTURE_TYPE_GENERATED_COMMANDS_INFO_NV**

- VUID-VkGeneratedCommandsInfoNV-pNext-pNext

  **pNext** **must** be **NULL**

- VUID-VkGeneratedCommandsInfoNV-pipelineBindPoint-parameter

  **pipelineBindPoint** **must** be a valid **VkPipelineBindPoint** value

- VUID-VkGeneratedCommandsInfoNV-pipeline-parameter

  **pipeline** **must** be a valid **VkPipeline** handle

- VUID-VkGeneratedCommandsInfoNV-indirectCommandsLayout-parameter

  **indirectCommandsLayout** **must** be a valid **VkIndirectCommandsLayoutNV** handle

- VUID-VkGeneratedCommandsInfoNV-pStreams-parameter

  **pStreams** **must** be a valid pointer to an array of **streamCount** valid
  **VkIndirectCommandsStreamNV** structures

- VUID-VkGeneratedCommandsInfoNV-preprocessBuffer-parameter

  **preprocessBuffer** **must** be a valid **VkBuffer** handle

- VUID-VkGeneratedCommandsInfoNV-sequencesCountBuffer-parameter

  If **sequencesCountBuffer** is not **VK_NULL_HANDLE**, **sequencesCountBuffer** **must** be a valid
  **VkBuffer** handle

- VUID-VkGeneratedCommandsInfoNV-sequencesIndexBuffer-parameter

  If **sequencesIndexBuffer** is not **VK_NULL_HANDLE**, **sequencesIndexBuffer** **must** be a valid
  **VkBuffer** handle

- VUID-VkGeneratedCommandsInfoNV-streamCount-arraylength

  **streamCount** **must** be greater than **0**

- VUID-VkGeneratedCommandsInfoNV-commonparent

  Each of **indirectCommandsLayout**, **pipeline**, **preprocessBuffer**, **sequencesCountBuffer**, and
  **sequencesIndexBuffer** that are valid handles of non-ignored parameters **must** have been
  created, allocated, or retrieved from the same **VkDevice**

Referencing the functions defined in **Indirect Commands Layout**, **vkCmdExecuteGeneratedCommandsNV**
behaves as:
uint32_t sequencesCount = sequencesCountBuffer ?
    min(maxSequencesCount, sequencesCountBuffer.load_uint32(sequencesCountOffset)) :
    maxSequencesCount;

cmdProcessAllSequences(commandBuffer, pipeline,
    indirectCommandsLayout, pIndirectCommandsStreams,
    sequencesCount,
    sequencesIndexBuffer, sequencesIndexOffset);

// The stateful commands within indirectCommandsLayout will not
// affect the state of subsequent commands in the target
// command buffer (cmd)

Note
It is important to note that the values of all state related to the pipelineBindPoint used are undefined after this command.

Commands can be preprocessed prior execution using the following command:

```c
// Provided by VK_NV_device_generated_commands
void vkCmdPreprocessGeneratedCommandsNV(
    VkCommandBuffer commandBuffer,
    const VkGeneratedCommandsInfoNV* pGeneratedCommandsInfo);
```

- `commandBuffer` is the command buffer which does the preprocessing.
- `pGeneratedCommandsInfo` is a pointer to an instance of the `VkGeneratedCommandsInfoNV` structure containing parameters affecting the preprocessing step.

Valid Usage

- VUID-vkCmdPreprocessGeneratedCommandsNV-commandBuffer-02974
  `commandBuffer` must not be a protected command buffer
- VUID-vkCmdPreprocessGeneratedCommandsNV-pGeneratedCommandsInfo-02927
  `pGeneratedCommandsInfo`'s `indirectCommandsLayout` must have been created with the `VK_INDIRECT_COMMANDS_LAYOUT_USAGE_EXPLICIT_PREPROCESS_BIT_NV` bit set
- VUID-vkCmdPreprocessGeneratedCommandsNV-deviceGeneratedCommands-02928
  The `VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV::deviceGeneratedCommands` feature must be enabled
Valid Usage (Implicit)

- VUID-vkCmdPreprocessGeneratedCommandsNV-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdPreprocessGeneratedCommandsNV-pGeneratedCommandsInfo-parameter
  
  `pGeneratedCommandsInfo` must be a valid pointer to a valid `VkGeneratedCommandsInfoNV` structure

- VUID-vkCmdPreprocessGeneratedCommandsNV-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdPreprocessGeneratedCommandsNV-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- VUID-vkCmdPreprocessGeneratedCommandsNV-renderpass
  
  This command must only be called outside of a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 32. Sparse Resources

As documented in Resource Memory Association, VkBuffer and VkImage resources in Vulkan must be bound completely and contiguously to a single VkDeviceMemory object. This binding must be done before the resource is used, and the binding is immutable for the lifetime of the resource.

Sparse resources relax these restrictions and provide these additional features:

- Sparse resources can be bound non-contiguously to one or more VkDeviceMemory allocations.
- Sparse resources can be re-bound to different memory allocations over the lifetime of the resource.
- Sparse resources can have descriptors generated and used orthogonally with memory binding commands.

32.1. Sparse Resource Features

Sparse resources have several features that must be enabled explicitly at resource creation time. The features are enabled by including bits in the flags parameter of VkImageCreateInfo or VkBufferCreateInfo. Each feature also has one or more corresponding feature enables specified in VkPhysicalDeviceFeatures.

- **Sparse binding** is the base feature, and provides the following capabilities:
  - Resources can be bound at some defined (sparse block) granularity.
  - The entire resource must be bound to memory before use regardless of regions actually accessed.
  - No specific mapping of image region to memory offset is defined, i.e. the location that each texel corresponds to in memory is implementation-dependent.
  - Sparse buffers have a well-defined mapping of buffer range to memory range, where an offset into a range of the buffer that is bound to a single contiguous range of memory corresponds to an identical offset within that range of memory.
  - Requested via the VK_IMAGE_CREATE_SPARSE_BINDING_BIT and VK_BUFFER_CREATE_SPARSE_BINDING_BIT bits.
  - A sparse image created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT (but not VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT) supports all formats that non-sparse usage supports, and supports both VK_IMAGE_TILING_OPTIMAL and VK_IMAGE_TILING_LINEAR tiling.

- **Sparse Residency** builds on (and requires) the sparseBinding feature. It includes the following capabilities:
  - Resources do not have to be completely bound to memory before use on the device.
  - Images have a prescribed sparse image block layout, allowing specific rectangular regions of the image to be bound to specific offsets in memory allocations.
  - Consistency of access to unbound regions of the resource is defined by the absence or presence of VkPhysicalDeviceSparseProperties::residencyNonResidentStrict. If this property is present, accesses to unbound regions of the resource are well defined and behave as if the
data bound is populated with all zeros; writes are discarded. When this property is absent, accesses are considered safe, but reads will return undefined values.

- Requested via the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` and `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bits.

- Sparse residency support is advertised on a finer grain via the following features:
  - `sparseResidencyBuffer`: Support for creating `VkBuffer` objects with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidencyImage2D`: Support for creating 2D single-sampled `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidencyImage3D`: Support for creating 3D `VkImage` objects with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency2Samples`: Support for creating 2D `VkImage` objects with 2 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency4Samples`: Support for creating 2D `VkImage` objects with 4 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency8Samples`: Support for creating 2D `VkImage` objects with 8 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.
  - `sparseResidency16Samples`: Support for creating 2D `VkImage` objects with 16 samples and `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

Implementations supporting `sparseResidencyImage2D` are only required to support sparse 2D, single-sampled images. Support for sparse 3D and MSAA images is optional and can be enabled via `sparseResidencyImage3D`, `sparseResidency2Samples`, `sparseResidency4Samples`, `sparseResidency8Samples`, and `sparseResidency16Samples`.

- A sparse image created using `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` supports all non-compressed color formats with power-of-two element size that non-sparse usage supports. Additional formats may also be supported and can be queried via `vkGetPhysicalDeviceSparseImageFormatProperties`. `VK_IMAGE_TILING_LINEAR` tiling is not supported.

- Sparse aliasing provides the following capability that can be enabled per resource:
  Allows physical memory ranges to be shared between multiple locations in the same sparse resource or between multiple sparse resources, with each binding of a memory location observing a consistent interpretation of the memory contents.

  See Sparse Memory Aliasing for more information.

### 32.2. Sparse Buffers and Fully-Resident Images

Both `VkBuffer` and `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_BINDING_BIT` or `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` bits can be thought of as a linear region of address space. In the `VkImage` case if `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` is not used, this linear region is entirely opaque, meaning that there is no application-visible mapping between texel location and memory.
Unless `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` or `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` are also used, the entire resource must be bound to one or more `VkDeviceMemory` objects before use.

### 32.2.1. Sparse Buffer and Fully-Resident Image Block Size

The sparse block size in bytes for sparse buffers and fully-resident images is reported as `VkMemoryRequirements::alignment`. `alignment` represents both the memory alignment requirement and the binding granularity (in bytes) for sparse resources.

### 32.3. Sparse Partially-Resident Buffers

`VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` bit allow the buffer to be made only partially resident. Partially resident `VkBuffer` objects are allocated and bound identically to `VkBuffer` objects using only the `VK_BUFFER_CREATE_SPARSE_BINDING_BIT` feature. The only difference is the ability for some regions of the buffer to be unbound during device use.

### 32.4. Sparse Partially-Resident Images

`VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` bit allow specific rectangular regions of the image called sparse image blocks to be bound to specific ranges of memory. This allows the application to manage residency at either image subresource or sparse image block granularity. Each image subresource (outside of the `mip tail`) starts on a sparse block boundary and has dimensions that are integer multiples of the corresponding dimensions of the sparse image block.

**Note**

Applications can use these types of images to control LOD based on total memory consumption. If memory pressure becomes an issue the application can unbind and disable specific mipmap levels of images without having to recreate resources or modify texel data of unaffected levels.

The application can also use this functionality to access subregions of the image in a “megatexture” fashion. The application can create a large image and only populate the region of the image that is currently being used in the scene.

### 32.4.1. Accessing Unbound Regions

The following member of `VkPhysicalDeviceSparseProperties` affects how data in unbound regions of sparse resources are handled by the implementation:

- `residencyNonResidentStrict`

If this property is not present, reads of unbound regions of the image will return undefined values. Both reads and writes are still considered safe and will not affect other resources or populated regions of the image.
If this property is present, all reads of unbound regions of the image will behave as if the region was bound to memory populated with all zeros; writes will be discarded.

Formatted accesses to unbound memory may still alter some component values in the natural way for those accesses, e.g. substituting a value of one for alpha in formats that do not have an alpha component.

Example: Reading the alpha component of an unbacked VK_FORMAT_R8_UNORM image will return a value of 1.0f.

See Physical Device Enumeration for instructions for retrieving physical device properties.

Implementor’s Note

For implementations that cannot natively handle access to unbound regions of a resource, the implementation may allocate and bind memory to the unbound regions. Reads and writes to unbound regions will access the implementation-managed memory instead.

Given that the values resulting from reads of unbound regions are undefined in this scenario, implementations may use the same physical memory for all unbound regions of multiple resources within the same process.

32.4.2. Mip Tail Regions

Sparse images created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT (without also using VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT) have no specific mapping of image region or image subresource to memory offset defined, so the entire image can be thought of as a linear opaque address region. However, images created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT do have a prescribed sparse image block layout, and hence each image subresource must start on a sparse block boundary. Within each array layer, the set of mip levels that have a smaller size than the sparse block size in bytes are grouped together into a mip tail region.

If the VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT flag is present in the flags member of VkSparseImageFormatProperties, for the image's format, then any mip level which has dimensions that are not integer multiples of the corresponding dimensions of the sparse image block, and all subsequent mip levels, are also included in the mip tail region.

The following member of VkPhysicalDeviceSparseProperties may affect how the implementation places mip levels in the mip tail region:

- residencyAlignedMipSize

Each mip tail region is bound to memory as an opaque region (i.e. must be bound using a VkSparseImageOpaqueMemoryBindInfo structure) and may be of a size greater than or equal to the sparse block size in bytes. This size is guaranteed to be an integer multiple of the sparse block size in bytes.
An implementation may choose to allow each array-layer's mip tail region to be bound to memory independently or require that all array-layer's mip tail regions be treated as one. This is dictated by `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` in `VkSparseImageMemoryRequirements::flags`.

The following diagrams depict how `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` alter memory usage and requirements.

![Figure 19. Sparse Image](image_url)

In the absence of `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, each array layer contains a mip tail region containing texel data for all mip levels smaller than the sparse image block in any dimension.

Mip levels that are as large or larger than a sparse image block in all dimensions can be bound individually. Right-edges and bottom-edges of each level are allowed to have partially used sparse blocks. Any bound partially-used-sparse-blocks must still have their full sparse block size in bytes allocated in memory.
When `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` is present all array layers will share a single mip tail region.

**Note**

The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.
When `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` is present the first mip level that would contain partially used sparse blocks begins the mip tail region. This level and all subsequent levels are placed in the mip tail. Only the first N mip levels whose dimensions are an exact multiple of the sparse image block dimensions can be bound and unbound on a sparse block basis.

![Figure 22. Sparse Image with Aligned Mip Size and Single Mip Tail](image)

**Note**

The mip tail region is presented here in a 2D array simply for figure size reasons. It is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

When both `VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT` and `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` are present the constraints from each of these flags are in effect.

### 32.4.3. Standard Sparse Image Block Shapes

Standard sparse image block shapes define a standard set of dimensions for sparse image blocks that depend on the format of the image. Layout of texels or compressed texel blocks within a sparse image block is implementation dependent. All currently defined standard sparse image block shapes are 64 KB in size.

For block-compressed formats (e.g. `VK_FORMAT_BC5_UNORM_BLOCK`), the texel size is the size of the compressed texel block (e.g. 128-bit for BC5) thus the dimensions of the standard sparse image block shapes apply in terms of compressed texel blocks.
Note

For block-compressed formats, the dimensions of a sparse image block in terms of texels can be calculated by multiplying the sparse image block dimensions by the compressed texel block dimensions.
Table 46. Standard Sparse Image Block Shapes (Single Sample)

<table>
<thead>
<tr>
<th>TEXEL SIZE (bits)</th>
<th>Block Shape (2D)</th>
<th>Block Shape (3D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit</td>
<td>256 × 256 × 1</td>
<td>64 × 32 × 32</td>
</tr>
<tr>
<td>16-Bit</td>
<td>256 × 128 × 1</td>
<td>32 × 32 × 32</td>
</tr>
<tr>
<td>32-Bit</td>
<td>128 × 128 × 1</td>
<td>32 × 32 × 16</td>
</tr>
<tr>
<td>64-Bit</td>
<td>128 × 64 × 1</td>
<td>32 × 16 × 16</td>
</tr>
<tr>
<td>128-Bit</td>
<td>64 × 64 × 1</td>
<td>16 × 16 × 16</td>
</tr>
</tbody>
</table>

Table 47. Standard Sparse Image Block Shapes (MSAA)

<table>
<thead>
<tr>
<th>TEXEL SIZE (bits)</th>
<th>Block Shape (2X)</th>
<th>Block Shape (4X)</th>
<th>Block Shape (8X)</th>
<th>Block Shape (16X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit</td>
<td>128 × 256 × 1</td>
<td>128 × 128 × 1</td>
<td>64 × 128 × 1</td>
<td>64 × 64 × 1</td>
</tr>
<tr>
<td>16-Bit</td>
<td>128 × 128 × 1</td>
<td>128 × 64 × 1</td>
<td>64 × 64 × 1</td>
<td>64 × 32 × 1</td>
</tr>
<tr>
<td>32-Bit</td>
<td>64 × 128 × 1</td>
<td>64 × 64 × 1</td>
<td>32 × 64 × 1</td>
<td>32 × 32 × 1</td>
</tr>
<tr>
<td>64-Bit</td>
<td>64 × 64 × 1</td>
<td>64 × 32 × 1</td>
<td>32 × 32 × 1</td>
<td>32 × 16 × 1</td>
</tr>
<tr>
<td>128-Bit</td>
<td>32 × 64 × 1</td>
<td>32 × 32 × 1</td>
<td>16 × 32 × 1</td>
<td>16 × 16 × 1</td>
</tr>
</tbody>
</table>

Implementations that support the standard sparse image block shape for all formats listed in the Standard Sparse Image Block Shapes (Single Sample) and Standard Sparse Image Block Shapes (MSAA) tables may advertise the following VkPhysicalDeviceSparseProperties:

- residencyStandard2DBlockShape
- residencyStandard2DMultisampleBlockShape
- residencyStandard3DBlockShape

Reporting each of these features does not imply that all possible image types are supported as sparse. Instead, this indicates that no supported sparse image of the corresponding type will use custom sparse image block dimensions for any formats that have a corresponding standard sparse image block shape.

### 32.4.4. Custom Sparse Image Block Shapes

An implementation that does not support a standard image block shape for a particular sparse partially-resident image may choose to support a custom sparse image block shape for it instead. The dimensions of such a custom sparse image block shape are reported in VkSparseImageFormatProperties::imageGranularity. As with standard sparse image block shapes, the size in bytes of the custom sparse image block shape will be reported in VkMemoryRequirements::alignment.

Custom sparse image block dimensions are reported through vkGetPhysicalDeviceSparseImageFormatProperties and vkGetImageSparseMemoryRequirements.

An implementation must not support both the standard sparse image block shape and a custom
sparse image block shape for the same image. The standard sparse image block shape must be used if it is supported.

32.4.5. Multiple Aspects

Partially resident images are allowed to report separate sparse properties for different aspects of the image. One example is for depth/stencil images where the implementation separates the depth and stencil data into separate planes. Another reason for multiple aspects is to allow the application to manage memory allocation for implementation-private metadata associated with the image. See the figure below:

![Multiple Aspect Sparse Image](image)

**Figure 23. Multiple Aspect Sparse Image**

*Note*

The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of sparse blocks with an implementation-dependent mapping of texels or compressed texel blocks to sparse blocks.

In the figure above the depth, stencil, and metadata aspects all have unique sparse properties. The per-texel stencil data is \(\frac{1}{4}\) the size of the depth data, hence the stencil sparse blocks include \(4 \times\) the number of texels. The sparse block size in bytes for all of the aspects is identical and defined by `VkMemoryRequirements::alignment`.

**Metadata**

The metadata aspect of an image has the following constraints:
• All metadata is reported in the mip tail region of the metadata aspect.
• All metadata must be bound prior to device use of the sparse image.

32.5. Sparse Memory Aliasing

By default sparse resources have the same aliasing rules as non-sparse resources. See Memory Aliasing for more information.

VkDevice objects that have the sparseResidencyAliased feature enabled are able to use the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT flags for resource creation. These flags allow resources to access physical memory bound into multiple locations within one or more sparse resources in a data consistent fashion. This means that reading physical memory from multiple aliased locations will return the same value.

Care must be taken when performing a write operation to aliased physical memory. Memory dependencies must be used to separate writes to one alias from reads or writes to another alias. Writes to aliased memory that are not properly guarded against accesses to different aliases will have undefined results for all accesses to the aliased memory.

Applications that wish to make use of data consistent sparse memory aliasing must abide by the following guidelines:

• All sparse resources that are bound to aliased physical memory must be created with the VK_BUFFER_CREATE_SPARSE_ALIASED.Bit / VK_IMAGE_CREATE_SPARSE_ALIASED.Bit flag.
• All resources that access aliased physical memory must interpret the memory in the same way. This implies the following:
  ◦ Buffers and images cannot alias the same physical memory in a data consistent fashion. The physical memory ranges must be used exclusively by buffers or used exclusively by images for data consistency to be guaranteed.
  ◦ Memory in sparse image mip tail regions cannot access aliased memory in a data consistent fashion.
  ◦ Sparse images that alias the same physical memory must have compatible formats and be using the same sparse image block shape in order to access aliased memory in a data consistent fashion.

Failure to follow any of the above guidelines will require the application to abide by the normal, non-sparse resource aliasing rules. In this case memory cannot be accessed in a data consistent fashion.

Note
Enabling sparse resource memory aliasing can be a way to lower physical memory use, but it may reduce performance on some implementations. An application developer can test on their target HW and balance the memory / performance trade-offs measured.
32.6. Sparse Resource Implementation Guidelines (Informative)
This section is Informative. It is included to aid in implementors’ understanding of sparse resources.

**Device Virtual Address**

The basic `sparseBinding` feature allows the resource to reserve its own device virtual address range at resource creation time rather than relying on a bind operation to set this. Without any other creation flags, no other constraints are relaxed compared to normal resources. All pages **must** be bound to physical memory before the device accesses the resource.

The `sparse residency` features allow sparse resources to be used even when not all pages are bound to memory. Implementations that support access to unbound pages without causing a fault **may** support `residencyNonResidentStrict`.

Not faulting on access to unbound pages is not enough to support `residencyNonResidentStrict`. An implementation **must** also guarantee that reads after writes to unbound regions of the resource always return data for the read as if the memory contains zeros. Depending on any caching hierarchy of the implementation this **may** not always be possible.

Any implementation that does not fault, but does not guarantee correct read values **must** not support `residencyNonResidentStrict`.

Any implementation that **cannot** access unbound pages without causing a fault will require the implementation to bind the entire device virtual address range to physical memory. Any pages that the application does not bind to memory **may** be bound to one (or more) "`placeholder" physical page(s) allocated by the implementation. Given the following properties:

- A process **must** not access memory from another process
- Reads return undefined values

It is sufficient for each host process to allocate these placeholder pages and use them for all resources in that process. Implementations **may** allocate more often (per instance, per device, or per resource).

**Binding Memory**

The byte size reported in `VkMemoryRequirements::size` **must** be greater than or equal to the amount of physical memory **required** to fully populate the resource. Some implementations require “holes” in the device virtual address range that are never accessed. These holes **may** be included in the `size` reported for the resource.

Including or not including the device virtual address holes in the resource size will alter how the implementation provides support for `VkSparseImageOpaqueMemoryBindInfo`. This operation **must** be supported for all sparse images, even ones created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`.

- If the holes are included in the size, this bind function becomes very easy. In most cases the `resourceOffset` is simply a device virtual address offset and the implementation can easily determine what device virtual address to bind. The cost is that the application **may**
allocate more physical memory for the resource than it needs.

- If the holes are not included in the size, the application can allocate less physical memory than otherwise for the resource. However, in this case the implementation must account for the holes when mapping resourceOffset to the actual device virtual address intended to be mapped.

**Note**

If the application always uses `VkSparseImageMemoryBindInfo` to bind memory for the non-tail mip levels, any holes that are present in the resource size may never be bound.

Since `VkSparseImageMemoryBindInfo` uses texel locations to determine which device virtual addresses to bind, it is impossible to bind device virtual address holes with this operation.

**Binding Metadata Memory**

All metadata for sparse images have their own sparse properties and are embedded in the mip tail region for said properties. See the Multiaspect section for details.

Given that metadata is in a mip tail region, and the mip tail region must be reported as contiguous (either globally or per-array-layer), some implementations will have to resort to complicated offset \(\rightarrow\) device virtual address mapping for handling `VkSparseImageOpaqueMemoryBindInfo`.

To make this easier on the implementation, the `VK_SPARSE_MEMORY_BIND_METADATA_BIT` explicitly specifies when metadata is bound with `VkSparseImageOpaqueMemoryBindInfo`. When this flag is not present, the resourceOffset may be treated as a strict device virtual address offset.

When `VK_SPARSE_MEMORY_BIND_METADATA_BIT` is present, the resourceOffset must have been derived explicitly from the imageMipTailOffset in the sparse resource properties returned for the metadata aspect. By manipulating the value returned for imageMipTailOffset, the resourceOffset does not have to correlate directly to a device virtual address offset, and may instead be whatever value makes it easiest for the implementation to derive the correct device virtual address.

### 32.7. Sparse Resource API

The APIs related to sparse resources are grouped into the following categories:

- Physical Device Features
- Physical Device Sparse Properties
- Sparse Image Format Properties
- Sparse Resource Creation
- Sparse Resource Memory Requirements
- Binding Resource Memory
32.7.1. Physical Device Features

Some sparse-resource related features are reported and enabled in VkPhysicalDeviceFeatures. These features must be supported and enabled on the VkDevice object before applications can use them. See Physical Device Features for information on how to get and set enabled device features, and for more detailed explanations of these features.

Sparse Physical Device Features

- sparseBinding: Support for creating VkBuffer and VkImage objects with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT and VK_IMAGE_CREATE_SPARSE_BINDING_BIT flags, respectively.
- sparseResidencyBuffer: Support for creating VkBuffer objects with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag.
- sparseResidencyImage2D: Support for creating 2D single-sampled VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- sparseResidencyImage3D: Support for creating 3D VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- sparseResidency2Samples: Support for creating 2D VkImage objects with 2 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- sparseResidency4Samples: Support for creating 2D VkImage objects with 4 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- sparseResidency8Samples: Support for creating 2D VkImage objects with 8 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- sparseResidency16Samples: Support for creating 2D VkImage objects with 16 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- sparseResidencyAliased: Support for creating VkBuffer and VkImage objects with the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT flags, respectively.

32.7.2. Physical Device Sparse Properties

Some features of the implementation are not possible to disable, and are reported to allow applications to alter their sparse resource usage accordingly. These read-only capabilities are reported in the VkPhysicalDeviceProperties::sparseProperties member, which is a structure of type VkPhysicalDeviceSparseProperties.

The VkPhysicalDeviceSparseProperties structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceSparseProperties {
    VkBool32 residencyStandard2DBlockShape;
    VkBool32 residencyStandard2DMultisampleBlockShape;
    VkBool32 residencyStandard3DBlockShape;
    VkBool32 residencyAlignedMipSize;
    VkBool32 residencyNonResidentStrict;
} VkPhysicalDeviceSparseProperties;
```
residencyStandard2DBlockShape is VK_TRUE if the physical device will access all single-sample 2D sparse resources using the standard sparse image block shapes (based on image format), as described in the Standard Sparse Image Block Shapes (Single Sample) table. If this property is not supported the value returned in the imageGranularity member of the VkSparseImageFormatProperties structure for single-sample 2D images is not required to match the standard sparse image block dimensions listed in the table.

residencyStandard2DMultisampleBlockShape is VK_TRUE if the physical device will access all multisample 2D sparse resources using the standard sparse image block shapes (based on image format), as described in the Standard Sparse Image Block Shapes (MSAA) table. If this property is not supported, the value returned in the imageGranularity member of the VkSparseImageFormatProperties structure for multisample 2D images is not required to match the standard sparse image block dimensions listed in the table.

residencyStandard3DBlockShape is VK_TRUE if the physical device will access all 3D sparse resources using the standard sparse image block shapes (based on image format), as described in the Standard Sparse Image Block Shapes (Single Sample) table. If this property is not supported, the value returned in the imageGranularity member of the VkSparseImageFormatProperties structure for 3D images is not required to match the standard sparse image block dimensions listed in the table.

residencyAlignedMipSize is VK_TRUE if images with mip level dimensions that are not integer multiples of the corresponding dimensions of the sparse image block may be placed in the mip tail. If this property is not reported, only mip levels with dimensions smaller than the imageGranularity member of the VkSparseImageFormatProperties structure will be placed in the mip tail. If this property is reported the implementation is allowed to return VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT in the flags member of VkSparseImageFormatProperties, indicating that mip level dimensions that are not integer multiples of the corresponding dimensions of the sparse image block will be placed in the mip tail.

residencyNonResidentStrict specifies whether the physical device can consistently access non-resident regions of a resource. If this property is VK_TRUE, access to non-resident regions of resources will be guaranteed to return values as if the resource was populated with 0; writes to non-resident regions will be discarded.

32.7.3. Sparse Image Format Properties

Given that certain aspects of sparse image support, including the sparse image block dimensions, may be implementation-dependent, VkGetPhysicalDeviceSparseImageFormatProperties can be used to query for sparse image format properties prior to resource creation. This command is used to check whether a given set of sparse image parameters is supported and what the sparse image block shape will be.

Sparse Image Format Properties API

The VkSparseImageFormatProperties structure is defined as:
typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags          aspectMask;
    VkExtent3D                  imageGranularity;
    VkSparseImageFormatFlags    flags;
} VkSparseImageFormatProperties;

- **aspectMask** is a bitmask `VkImageAspectFlagBits` specifying which aspects of the image the properties apply to.
- **imageGranularity** is the width, height, and depth of the sparse image block in texels or compressed texel blocks.
- **flags** is a bitmask of `VkSparseImageFormatFlagBits` specifying additional information about the sparse resource.

Bits which *may* be set in `VkSparseImageFormatProperties::flags`, specifying additional information about the sparse resource, are:

```
// Provided by VK_VERSION_1_0
typedef enum VkSparseImageFormatFlagBits {
    VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT = 0x00000001,
    VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT = 0x00000002,
    VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT = 0x00000004,
} VkSparseImageFormatFlagBits;
```

- **VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT** specifies that the image uses a single mip tail region for all array layers.
- **VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT** specifies that the first mip level whose dimensions are not integer multiples of the corresponding dimensions of the sparse image block begins the mip tail region.
- **VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT** specifies that the image uses non-standard sparse image block dimensions, and the `imageGranularity` values do not match the standard sparse image block dimensions for the given format.

```
// Provided by VK_VERSION_1_0
typedef VkFlags VkSparseImageFormatFlags;
```

`VkSparseImageFormatFlags` is a bitmask type for setting a mask of zero or more `VkSparseImageFormatFlagBits`.

`vkGetPhysicalDeviceSparseImageFormatProperties` returns an array of `VkSparseImageFormatProperties`. Each element will describe properties for one set of image aspects that are bound simultaneously in the image. This is usually one element for each aspect in the image, but for interleaved depth/stencil images there is only one element describing the combined aspects.
void vkGetPhysicalDeviceSparseImageFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkImageType type,
    VkSampleCountFlagBits samples,
    VkImageUsageFlags usage,
    VkImageTiling tiling,
    uint32_t* pPropertyCount,
    VkSparseImageFormatProperties* pProperties);

- **physicalDevice** is the physical device from which to query the sparse image format properties.
- **format** is the image format.
- **type** is the dimensionality of image.
- **samples** is a VkSampleCountFlagBits value specifying the number of samples per texel.
- **usage** is a bitmask describing the intended usage of the image.
- **tiling** is the tiling arrangement of the texel blocks in memory.
- **pPropertyCount** is a pointer to an integer related to the number of sparse format properties available or queried, as described below.
- **pProperties** is either NULL or a pointer to an array of VkSparseImageFormatProperties structures.

If **pProperties** is NULL, then the number of sparse format properties available is returned in **pPropertyCount**. Otherwise, **pPropertyCount** must point to a variable set by the user to the number of elements in the **pProperties** array, and on return the variable is overwritten with the number of structures actually written to **pProperties**. If **pPropertyCount** is less than the number of sparse format properties available, at most **pPropertyCount** structures will be written.

If VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT is not supported for the given arguments, **pPropertyCount** will be set to zero upon return, and no data will be written to **pProperties**.

Multiple aspects are returned for depth/stencil images that are implemented as separate planes by the implementation. The depth and stencil data planes each have unique VkSparseImageFormatProperties data.

Depth/stencil images with depth and stencil data interleaved into a single plane will return a single VkSparseImageFormatProperties structure with the aspectMask set to VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT.
Valid Usage

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-samples-01094
  samples must be a bit value that is set in VkImageFormatProperties::sampleCounts returned by vkGetPhysicalDeviceImageFormatProperties with format, type, tiling, and usage equal to those in this command and flags equal to the value that is set in VkImageCreateInfo ::flags when the image is created

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-format-parameter
  format must be a valid VkFormat value

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-type-parameter
  type must be a valid VkImageType value

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-samples-parameter
  samples must be a valid VkSampleCountFlagBits value

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-usage-parameter
  usage must be a valid combination of VkImageUsageFlagBits values

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-usage-requiredbitmask
  usage must not be 0

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-tiling-parameter
  tiling must be a valid VkImageTiling value

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties-pPropertyCount-parameter
  pPropertyCount must be a valid pointer to a uint32_t value

  If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties must be a valid pointer to an array of pPropertyCount VkSparseImageFormatProperties structures

vkGetPhysicalDeviceSparseImageFormatProperties2 returns an array of VkSparseImageFormatProperties2. Each element will describe properties for one set of image aspects that are bound simultaneously in the image. This is usually one element for each aspect in the image, but for interleaved depth/stencil images there is only one element describing the combined aspects.
void vkGetPhysicalDeviceSparseImageFormatProperties2(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceSparseImageFormatInfo2* pFormatInfo,
    uint32_t* pPropertyCount,
    VkSparseImageFormatProperties2* pProperties);

or the equivalent command

void vkGetPhysicalDeviceSparseImageFormatProperties2KHR(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceSparseImageFormatInfo2* pFormatInfo,
    uint32_t* pPropertyCount,
    VkSparseImageFormatProperties2* pProperties);

- physicalDevice is the physical device from which to query the sparse image format properties.
- pFormatInfo is a pointer to a VkPhysicalDeviceSparseImageFormatInfo2 structure containing input parameters to the command.
- pPropertyCount is a pointer to an integer related to the number of sparse format properties available or queried, as described below.
- pProperties is either NULL or a pointer to an array of VkSparseImageFormatProperties2 structures.

vkGetPhysicalDeviceSparseImageFormatProperties2 behaves identically to vkGetPhysicalDeviceSparseImageFormatProperties, with the ability to return extended information by adding extending structures to the pNext chain of its pProperties parameter.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-physicalDevice-parameter physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-pFormatInfo-parameter pFormatInfo must be a valid pointer to a valid VkPhysicalDeviceSparseImageFormatInfo2 structure
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-pPropertyCount-parameter pPropertyCount must be a valid pointer to a uint32_t value
- VUID-vkGetPhysicalDeviceSparseImageFormatProperties2-pProperties-parameter If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties must be a valid pointer to an array of pPropertyCount VkSparseImageFormatProperties2 structures

The VkPhysicalDeviceSparseImageFormatInfo2 structure is defined as:
typedef struct VkPhysicalDeviceSparseImageFormatInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkImageType type;
    VkSampleCountFlagBits samples;
    VkImageUsageFlags usage;
    VkImageTiling tiling;
} VkPhysicalDeviceSparseImageFormatInfo2;

or the equivalent

typedef VkPhysicalDeviceSparseImageFormatInfo2 VkPhysicalDeviceSparseImageFormatInfo2KHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **format** is the image format.
- **type** is the dimensionality of image.
- **samples** is the number of samples per texel as defined in **VkSampleCountFlagBits**.
- **usage** is a bitmask describing the intended usage of the image.
- **tiling** is the tiling arrangement of the texel blocks in memory.

Valid Usage

- VUID-VkPhysicalDeviceSparseImageFormatInfo2-samples-01095
  samples must be a bit value that is set in **VkImageFormatProperties::sampleCounts** returned by **vkGetPhysicalDeviceImageFormatProperties** with **format**, **type**, **tiling**, and **usage** equal to those in this command and **flags** equal to the value that is set in **VkImageCreateInfo::flags** when the image is created
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-sType-sType**
  
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2`

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-pNext-pNext**
  
  pNext must be NULL

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-format-parameter**
  
  format must be a valid `VkFormat` value

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-type-parameter**
  
  type must be a valid `VkImageType` value

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-samples-parameter**
  
  samples must be a valid `VkSampleCountFlagBits` value

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-usage-parameter**
  
  usage must be a valid combination of `VkImageUsageFlagBits` values

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-usage-requiredbitmask**
  
  usage must not be `0`

- **VUID-VkPhysicalDeviceSparseImageFormatInfo2-tiling-parameter**
  
  tiling must be a valid `VkImageTiling` value

The `VkSparseImageFormatProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSparseImageFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkSparseImageFormatProperties properties;
} VkSparseImageFormatProperties2;
```

or the equivalent

```c
// Provided by VK_KHR_get_physical_device_properties2
typedef VkSparseImageFormatProperties2 VkSparseImageFormatProperties2KHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **properties** is a `VkSparseImageFormatProperties` structure which is populated with the same values as in `vkGetPhysicalDeviceSparseImageFormatProperties`.
32.7.4. Sparse Resource Creation

Sparse resources require that one or more sparse feature flags be specified (as part of the VkPhysicalDeviceFeatures structure described previously in the Physical Device Features section) when calling vkCreateDevice. When the appropriate device features are enabled, the VK_BUFFER_CREATE_SPARSE_* and VK_IMAGE_CREATE_SPARSE_* flags can be used. See vkCreateBuffer and vkCreateImage for details of the resource creation APIs.

Note
Specifying VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT or VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT requires specifying VK_BUFFER_CREATE_SPARSE_BINDING_BIT or VK_IMAGE_CREATE_SPARSE_BINDING_BIT, respectively, as well. This means that resources must be created with the appropriate *_SPARSE_BINDING_BIT to be used with the sparse binding command (vkQueueBindSparse).

32.7.5. Sparse Resource Memory Requirements

Sparse resources have specific memory requirements related to binding sparse memory. These memory requirements are reported differently for VkBuffer objects and VkImage objects.

Buffer and Fully-Resident Images

Buffers (both fully and partially resident) and fully-resident images can be bound to memory using only the data from VkMemoryRequirements. For all sparse resources the VkMemoryRequirements::alignment member specifies both the bindable sparse block size in bytes and required alignment of VkDeviceMemory.

Partially Resident Images

Partially resident images have a different method for binding memory. As with buffers and fully resident images, the VkMemoryRequirements::alignment field specifies the bindable sparse block size in bytes for the image.

Requesting sparse memory requirements for VkImage objects using vkGetImageSparseMemoryRequirements will return an array of one or more VkSparseImageMemoryRequirements structures. Each structure describes the sparse memory requirements for a group of aspects of the image.

The sparse image must have been created using the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag to
retrieve valid sparse image memory requirements.

Sparse Image Memory Requirements

The `VkSparseImageMemoryRequirements` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageMemoryRequirements {
    VkSparseImageFormatProperties formatProperties;
    uint32_t imageMipTailFirstLod;
    VkDeviceSize imageMipTailSize;
    VkDeviceSize imageMipTailOffset;
    VkDeviceSize imageMipTailStride;
} VkSparseImageMemoryRequirements;
```

- `formatProperties.aspectMask` is the set of aspects of the image that this sparse memory requirement applies to. This will usually have a single aspect specified. However, depth/stencil images may have depth and stencil data interleaved in the same sparse block, in which case both `VK_IMAGE_ASPECT_DEPTH_BIT` and `VK_IMAGE_ASPECT_STENCIL_BIT` would be present.
- `formatProperties.imageGranularity` describes the dimensions of a single bindable sparse image block in texel units. For aspect `VK_IMAGE_ASPECT_METADATA_BIT`, all dimensions will be zero. All metadata is located in the mip tail region.
- `formatProperties.flags` is a bitmask of `VkSparseImageFormatFlagBits`:
  - If `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` is set the image uses a single mip tail region for all array layers.
  - If `VK_SPARSE_IMAGE_FORMATAligned_MIP_SIZE_BIT` is set the dimensions of mip levels must be integer multiples of the corresponding dimensions of the sparse image block for levels not located in the mip tail.
  - If `VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT` is set the image uses non-standard sparse image block dimensions. The `formatProperties.imageGranularity` values do not match the standard sparse image block dimension corresponding to the image's format.
- `imageMipTailFirstLod` is the first mip level at which image subresources are included in the mip tail region.
- `imageMipTailSize` is the memory size (in bytes) of the mip tail region. If `formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, this is the size of the whole mip tail, otherwise this is the size of the mip tail of a single array layer. This value is guaranteed to be a multiple of the sparse block size in bytes.
- `imageMipTailOffset` is the opaque memory offset used with `VkSparseImageOpaqueMemoryBindInfo` to bind the mip tail region(s).
- `imageMipTailStride` is the offset stride between each array-layer's mip tail, if `formatProperties.flags` does not contain `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` (otherwise the value is undefined).

To query sparse memory requirements for an image, call:
// Provided by VK_VERSION_1_0
void vkGetImageSparseMemoryRequirements(
    VkDevice device,
    VkImage image,
    uint32_t* pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements* pSparseMemoryRequirements);

- **device** is the logical device that owns the image.
- **image** is the `VkImage` object to get the memory requirements for.
- **pSparseMemoryRequirementCount** is a pointer to an integer related to the number of sparse memory requirements available or queried, as described below.
- **pSparseMemoryRequirements** is either `NULL` or a pointer to an array of `VkSparseImageMemoryRequirements` structures.

If `pSparseMemoryRequirements` is NULL, then the number of sparse memory requirements available is returned in `pSparseMemoryRequirementCount`. Otherwise, `pSparseMemoryRequirementCount` must point to a variable set by the user to the number of elements in the `pSparseMemoryRequirements` array, and on return the variable is overwritten with the number of structures actually written to `pSparseMemoryRequirements`. If `pSparseMemoryRequirementCount` is less than the number of sparse memory requirements available, at most `pSparseMemoryRequirementCount` structures will be written.

If the image was not created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` then `pSparseMemoryRequirementCount` will be set to zero and `pSparseMemoryRequirements` will not be written to.

**Note**

It is legal for an implementation to report a larger value in `VkMemoryRequirements::size` than would be obtained by adding together memory sizes for all `VkSparseImageMemoryRequirements` returned by `vkGetImageSparseMemoryRequirements`. This may occur when the implementation requires unused padding in the address range describing the resource.
Valid Usage (Implicit)

- VUID-vkGetImageSparseMemoryRequirements-device-parameter
device must be a valid VkDevice handle
- VUID-vkGetImageSparseMemoryRequirements-image-parameter
image must be a valid VkImage handle
- VUID-vkGetImageSparseMemoryRequirements-pSparseMemoryRequirementCount-parameter
pSparseMemoryRequirementCount must be a valid pointer to a uint32_t value
- VUID-vkGetImageSparseMemoryRequirements-pSparseMemoryRequirements-parameter
If the value referenced by pSparseMemoryRequirementCount is not 0, and pSparseMemoryRequirements is not NULL, pSparseMemoryRequirements must be a valid pointer to an array of pSparseMemoryRequirementCount VkSparseImageMemoryRequirements structures
- VUID-vkGetImageSparseMemoryRequirements-image-parent
image must have been created, allocated, or retrieved from device

To query sparse memory requirements for an image, call:

```c
// Provided by VK_VERSION_1_1
void vkGetImageSparseMemoryRequirements2(
    VkDevice                                    device,
    const VkImageSparseMemoryRequirementsInfo2* pInfo,
    uint32_t*                                   pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements2*           pSparseMemoryRequirements);
```

or the equivalent command

```c
// Provided by VK_KHR_get_memory_requirements2
void vkGetImageSparseMemoryRequirements2KHR(
    VkDevice                                    device,
    const VkImageSparseMemoryRequirementsInfo2* pInfo,
    uint32_t*                                   pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements2*           pSparseMemoryRequirements);
```

- device is the logical device that owns the image.
- pInfo is a pointer to a VkImageSparseMemoryRequirementsInfo2 structure containing parameters required for the memory requirements query.
- pSparseMemoryRequirementCount is a pointer to an integer related to the number of sparse memory requirements available or queried, as described below.
- pSparseMemoryRequirements is either NULL or a pointer to an array of VkSparseImageMemoryRequirements2 structures.
Valid Usage (Implicit)

- VUID-vkGetImageSparseMemoryRequirements2-device-parameter
device must be a valid VkDevice handle

- VUID-vkGetImageSparseMemoryRequirements2-pInfo-parameter
pInfo must be a valid pointer to a valid VkImageSparseMemoryRequirementsInfo2 structure

- VUID-vkGetImageSparseMemoryRequirements2-pSparseMemoryRequirementCount-parameter
pSparseMemoryRequirementCount must be a valid pointer to a uint32_t value

- VUID-vkGetImageSparseMemoryRequirements2-pSparseMemoryRequirements-parameter
If the value referenced by pSparseMemoryRequirementCount is not 0, and
pSparseMemoryRequirements is not NULL, pSparseMemoryRequirements must be a valid pointer
to an array of pSparseMemoryRequirementCount VkSparseImageMemoryRequirements2 structures

The VkImageSparseMemoryRequirementsInfo2 structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkImageSparseMemoryRequirementsInfo2 {
    VkStructureType sType;
    const void* pNext;
    VkImage image;
} VkImageSparseMemoryRequirementsInfo2;
```

or the equivalent

```c
// Provided by VK_KHR_get_memory_requirements2
typedef VkImageSparseMemoryRequirementsInfo2 VkImageSparseMemoryRequirementsInfo2KHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- image is the image to query.

Valid Usage (Implicit)

- VUID-VkImageSparseMemoryRequirementsInfo2-sType-sType
sType must be VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2

- VUID-VkImageSparseMemoryRequirementsInfo2-pNext-pNext
pNext must be NULL

- VUID-VkImageSparseMemoryRequirementsInfo2-image-parameter
image must be a valid VkImage handle
The `VkSparseImageMemoryRequirements2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSparseImageMemoryRequirements2 {
    VkStructureType sType;
    void* pNext;
    VkSparseImageMemoryRequirements memoryRequirements;
} VkSparseImageMemoryRequirements2;
```

or the equivalent

```c
// Provided by VK_KHR_get_memory_requirements2
typedef VkSparseImageMemoryRequirements2 VkSparseImageMemoryRequirements2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memoryRequirements` is a `VkSparseImageMemoryRequirements` structure describing the memory requirements of the sparse image.

**Valid Usage (Implicit)**

- `VUID-VkSparseImageMemoryRequirements2-sType-sType
  sType must be VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2`
- `VUID-VkSparseImageMemoryRequirements2-pNext-pNext
  pNext must be NULL`

### 32.7.6. Binding Resource Memory

Non-sparse resources are backed by a single physical allocation prior to device use (via `vkBindImageMemory` or `vkBindBufferMemory`), and their backing must not be changed. On the other hand, sparse resources can be bound to memory non-contiguously and these bindings can be altered during the lifetime of the resource.

**Note**

It is important to note that freeing a `VkDeviceMemory` object with `vkFreeMemory` will not cause resources (or resource regions) bound to the memory object to become unbound. Applications must not access resources bound to memory that has been freed.

Sparse memory bindings execute on a queue that includes the `VK_QUEUE_SPARSE_BINDING_BIT` bit. Applications must use synchronization primitives to guarantee that other queues do not access ranges of memory concurrently with a binding change. Applications can access other ranges of the same resource while a bind operation is executing.
Implementations **must** provide a guarantee that simultaneously binding sparse blocks while another queue accesses those same sparse blocks via a sparse resource **must** not access memory owned by another process or otherwise corrupt the system.

While some implementations **may** include `VK_QUEUE_SPARSE_BINDING_BIT` support in queue families that also include graphics and compute support, other implementations **may** only expose a `VK_QUEUE_SPARSE_BINDING_BIT`-only queue family. In either case, applications **must** use synchronization primitives to explicitly request any ordering dependencies between sparse memory binding operations and other graphics/compute/transfer operations, as sparse binding operations are not automatically ordered against command buffer execution, even within a single queue.

When binding memory explicitly for the `VK_IMAGE_ASPECT_METADATA_BIT` the application **must** use the `VK_SPARSE_MEMORY_BIND_METADATA_BIT` in the `VkSparseMemoryBind::flags` field when binding memory. Binding memory for metadata is done the same way as binding memory for the mip tail, with the addition of the `VK_SPARSE_MEMORY_BIND_METADATA_BIT` flag.

Binding the mip tail for any aspect **must** only be performed using `VkSparseImageOpaqueMemoryBindInfo`. If `formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`, then it **can** be bound with a single `VkSparseMemoryBind` structure, with `resourceOffset = imageMipTailOffset` and `size = imageMipTailSize`.

If `formatProperties.flags` does not contain `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT` then the offset for the mip tail in each array layer is given as:

```
arrayMipTailOffset = imageMipTailOffset + arrayLayer * imageMipTailStride;
```

and the mip tail **can** be bound with `layerCount` `VkSparseMemoryBind` structures, each using `size = imageMipTailSize` and `resourceOffset = arrayMipTailOffset` as defined above.

Sparse memory binding is handled by the following APIs and related data structures.

**Sparse Memory Binding Functions**

The `VkSparseMemoryBind` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseMemoryBind {
    VkDeviceSize               resourceOffset;
    VkDeviceSize               size;
    VkDeviceMemory             memory;
    VkDeviceSize               memoryOffset;
    VkSparseMemoryBindFlags    flags;
} VkSparseMemoryBind;
```
- `resourceOffset` is the offset into the resource.
- `size` is the size of the memory region to be bound.
- `memory` is the `VkDeviceMemory` object that the range of the resource is bound to. If `memory` is `VK_NULL_HANDLE`, the range is unbound.
- `memoryOffset` is the offset into the `VkDeviceMemory` object to bind the resource range to. If `memory` is `VK_NULL_HANDLE`, this value is ignored.
- `flags` is a bitmask of `VkSparseMemoryBindFlagBits` specifying usage of the binding operation.

The binding range \([resourceOffset, resourceOffset + size]\) has different constraints based on `flags`. If `flags` contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range must be within the mip tail region of the metadata aspect. This metadata region is defined by:

\[
\text{metadataRegion} = [\text{base}, \text{base} + \text{imageMipTailSize}]
\]

\[
\text{base} = \text{imageMipTailOffset} + \text{imageMipTailStride} \times n
\]

and `imageMipTailOffset`, `imageMipTailSize`, and `imageMipTailStride` values are from the `VkSparseImageMemoryRequirements` corresponding to the metadata aspect of the image, and `n` is a valid array layer index for the image,

`imageMipTailStride` is considered to be zero for aspects where `VkSparseImageMemoryRequirements::formatProperties.flags` contains `VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT`.

If `flags` does not contain `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range must be within the range \([0, \text{VkMemoryRequirements::size}]\).
Valid Usage

- VUID-VkSparseMemoryBind-memory-01096
  If `memory` is not `VK_NULL_HANDLE`, `memory` and `memoryOffset` must match the memory requirements of the resource, as described in section Resource Memory Association.

- VUID-VkSparseMemoryBind-memory-01097
  If `memory` is not `VK_NULL_HANDLE`, `memory` must not have been created with a memory type that reports `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` bit set.

- VUID-VkSparseMemoryBind-size-01098
  `size` must be greater than 0.

- VUID-VkSparseMemoryBind-resourceOffset-01099
  `resourceOffset` must be less than the size of the resource.

- VUID-VkSparseMemoryBind-size-01100
  `size` must be less than or equal to the size of the resource minus `resourceOffset`.

- VUID-VkSparseMemoryBind-memoryOffset-01101
  `memoryOffset` must be less than the size of `memory`.

- VUID-VkSparseMemoryBind-size-01102
  `size` must be less than or equal to the size of `memory` minus `memoryOffset`.

- VUID-VkSparseMemoryBind-memory-02730
  If `memory` was created with `VkExportMemoryAllocateInfo::handleTypes` not equal to 0, at least one handle type it contained must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` or `VkExternalMemoryImageCreateInfo::handleTypes` when the resource was created.

- VUID-VkSparseMemoryBind-memory-02731
  If `memory` was created by a memory import operation, the external handle type of the imported memory must also have been set in `VkExternalMemoryBufferCreateInfo::handleTypes` or `VkExternalMemoryImageCreateInfo::handleTypes` when the resource was created.

Valid Usage (Implicit)

- VUID-VkSparseMemoryBind-memory-parameter
  If `memory` is not `VK_NULL_HANDLE`, `memory` must be a valid `VkDeviceMemory` handle.

- VUID-VkSparseMemoryBind-flags-parameter
  `flags` must be a valid combination of `VkSparseMemoryBindFlagBits` values.

Bits which can be set in `VkSparseMemoryBind::flags`, specifying usage of a sparse memory binding operation, are:
typedef enum VkSparseMemoryBindFlagBits {
    VK_SPARSE_MEMORY_BIND_METADATA_BIT = 0x00000001,
} VkSparseMemoryBindFlagBits;

- **VK_SPARSE_MEMORY_BIND_METADATA_BIT** specifies that the memory being bound is only for the metadata aspect.

typedef VkFlags VkSparseMemoryBindFlags;

 VkSparseMemoryBindFlags is a bitmask type for setting a mask of zero or more VkSparseMemoryBindFlagBits.

Memory is bound to VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT flag using the following structure:

typedef struct VkSparseBufferMemoryBindInfo {
    VkBuffer                     buffer;
    uint32_t                     bindCount;
    const VkSparseMemoryBind*    pBinds;
} VkSparseBufferMemoryBindInfo;

- **buffer** is the VkBuffer object to be bound.
- **bindCount** is the number of VkSparseMemoryBind structures in the pBinds array.
- **pBinds** is a pointer to an array of VkSparseMemoryBind structures.

Valid Usage (Implicit)

- VUID-VkSparseBufferMemoryBindInfo-buffer-parameter
  buffer must be a valid VkBuffer handle

- VUID-VkSparseBufferMemoryBindInfo-pBinds-parameter
  pBinds must be a valid pointer to an array of bindCount valid VkSparseMemoryBind structures

- VUID-VkSparseBufferMemoryBindInfo-bindCount-arraylength
  bindCount must be greater than 0

Memory is bound to opaque regions of VkImage objects created with the VK_IMAGE_CREATE_SPARSE_BINDING_BIT flag using the following structure:
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageOpaqueMemoryBindInfo {
    VkImage                  image;
    uint32_t                 bindCount;
    const VkSparseMemoryBind* pBinds;
} VkSparseImageOpaqueMemoryBindInfo;

- **image** is the `VkImage` object to be bound.
- **bindCount** is the number of `VkSparseMemoryBind` structures in the `pBinds` array.
- **pBinds** is a pointer to an array of `VkSparseMemoryBind` structures.

## Valid Usage

- VUID-VkSparseImageOpaqueMemoryBindInfo-pBinds-01103
  If the *flags* member of any element of `pBinds` contains `VK_SPARSE_MEMORY_BIND_METADATA_BIT`, the binding range defined must be within the mip tail region of the metadata aspect of `image`.

## Valid Usage (Implicit)

- VUID-VkSparseImageOpaqueMemoryBindInfo-image-parameter
  **image** must be a valid `VkImage` handle

- VUID-VkSparseImageOpaqueMemoryBindInfo-pBinds-parameter
  **pBinds** must be a valid pointer to an array of `bindCount` valid `VkSparseMemoryBind` structures

- VUID-VkSparseImageOpaqueMemoryBindInfo-bindCount-arraylength
  **bindCount** must be greater than 0
Note

This operation is normally used to bind memory to fully-resident sparse images or for mip tail regions of partially resident images. However, it can also be used to bind memory for the entire binding range of partially resident images.

In case flags does not contain VK_SPARSE_MEMORY_BIND_METADATA_BIT, the resourceOffset is in the range [0, VkMemoryRequirements::size). This range includes data from all aspects of the image, including metadata. For most implementations this will probably mean that the resourceOffset is a simple device address offset within the resource. It is possible for an application to bind a range of memory that includes both resource data and metadata. However, the application would not know what part of the image the memory is used for, or if any range is being used for metadata.

When flags contains VK_SPARSE_MEMORY_BIND_METADATA_BIT, the binding range specified must be within the mip tail region of the metadata aspect. In this case the resourceOffset is not required to be a simple device address offset within the resource. However, it is defined to be within [imageMipTailOffset, imageMipTailOffset + imageMipTailSize) for the metadata aspect. See VkSparseMemoryBind for the full constraints on binding region with this flag present.

Memory can be bound to sparse image blocks of VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag using the following structure:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageMemoryBindInfo {
    VkImage           image;
    uint32_t          bindCount;
    const VkSparseImageMemoryBind* pBinds;
} VkSparseImageMemoryBindInfo;
```

- **image** is the VkImage object to be bound
- **bindCount** is the number of VkSparseImageMemoryBind structures in pBinds array
- **pBinds** is a pointer to an array of VkSparseImageMemoryBind structures
Valid Usage

- **VUID-VkSparseImageMemoryBindInfo-subresource-01722**
  The `subresource.mipLevel` member of each element of `pBinds` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created.

- **VUID-VkSparseImageMemoryBindInfo-subresource-01723**
  The `subresource.arrayLayer` member of each element of `pBinds` must be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created.

- **VUID-VkSparseImageMemoryBindInfo-image-02901**
  `image` must have been created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set.

Valid Usage (Implicit)

- **VUID-VkSparseImageMemoryBindInfo-image-parameter**
  `image` must be a valid `VkImage` handle.

- **VUID-VkSparseImageMemoryBindInfo-pBinds-parameter**
  `pBinds` must be a valid pointer to an array of `bindCount` valid `VkSparseImageMemoryBind` structures.

- **VUID-VkSparseImageMemoryBindInfo-bindCount-arraylength**
  `bindCount` must be greater than 0.

The `VkSparseImageMemoryBind` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkSparseImageMemoryBind {
    VkImageSubresource subresource;
    VkOffset3D          offset;
    VkExtent3D          extent;
    VkDeviceMemory      memory;
    VkDeviceSize        memoryOffset;
    VkSparseMemoryBindFlags flags;
} VkSparseImageMemoryBind;
```

- **subresource** is the image aspect and region of interest in the image.
- **offset** are the coordinates of the first texel within the image subresource to bind.
- **extent** is the size in texels of the region within the image subresource to bind. The extent must be a multiple of the sparse image block dimensions, except when binding sparse image blocks along the edge of an image subresource it can instead be such that any coordinate of `offset + extent` equals the corresponding dimensions of the image subresource.
- **memory** is the `VkDeviceMemory` object that the sparse image blocks of the image are bound to. If `memory` is `VK_NULL_HANDLE`, the sparse image blocks are unbound.
- **memoryOffset** is an offset into `VkDeviceMemory` object. If `memory` is `VK_NULL_HANDLE`, this value...
is ignored.

- **flags** are sparse memory binding flags.

## Valid Usage

- **VUID-VkSparseImageMemoryBind-memory-01104**
  If the **sparse aliased residency** feature is not enabled, and if any other resources are bound to ranges of **memory**, the range of **memory** being bound **must** not overlap with those bound ranges.

- **VUID-VkSparseImageMemoryBind-memory-01105**
  **memory** and **memoryOffset** **must** match the memory requirements of the calling command’s **image**, as described in section Resource Memory Association.

- **VUID-VkSparseImageMemoryBind-subresource-01106**
  **subresource** **must** be a valid image subresource for **image** (see Image Views).

- **VUID-VkSparseImageMemoryBind-offset-01107**
  **offset.x** **must** be a multiple of the sparse image block width (**VkSparseImageFormatProperties::imageGranularity::width**) of the image.

- **VUID-VkSparseImageMemoryBind-extent-01108**
  **extent.width** **must** either be a multiple of the sparse image block width of the image, or else \((\text{extent.width} + \text{offset.x})\) **must** equal the width of the image subresource.

- **VUID-VkSparseImageMemoryBind-offset-01109**
  **offset.y** **must** be a multiple of the sparse image block height (**VkSparseImageFormatProperties::imageGranularity::height**) of the image.

- **VUID-VkSparseImageMemoryBind-extent-01110**
  **extent.height** **must** either be a multiple of the sparse image block height of the image, or else \((\text{extent.height} + \text{offset.y})\) **must** equal the height of the image subresource.

- **VUID-VkSparseImageMemoryBind-offset-01111**
  **offset.z** **must** be a multiple of the sparse image block depth (**VkSparseImageFormatProperties::imageGranularity::depth**) of the image.

- **VUID-VkSparseImageMemoryBind-extent-01112**
  **extent.depth** **must** either be a multiple of the sparse image block depth of the image, or else \((\text{extent.depth} + \text{offset.z})\) **must** equal the depth of the image subresource.

- **VUID-VkSparseImageMemoryBind-memory-02732**
  If **memory** was created with **VkExportMemoryAllocateInfo::handleTypes** not equal to 0, at least one handle type it contained **must** also have been set in **VkExternalMemoryImageCreateInfo::handleTypes** when the image was created.

- **VUID-VkSparseImageMemoryBind-memory-02733**
  If **memory** was created by a memory import operation, the external handle type of the imported memory **must** also have been set in **VkExternalMemoryImageCreateInfo::handleTypes** when **image** was created.
Valid Usage (Implicit)

- VUID-VkSparseImageMemoryBind-subresource-parameter
  subresource must be a valid VkImageSubresource structure

- VUID-VkSparseImageMemoryBind-memory-parameter
  If memory is not VK_NULL_HANDLE, memory must be a valid VkDeviceMemory handle

- VUID-VkSparseImageMemoryBind-flags-parameter
  flags must be a valid combination of VkSparseMemoryBindFlagBits values

To submit sparse binding operations to a queue, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkQueueBindSparse(
    VkQueue                                 queue,
    uint32_t                                 bindInfoCount,
    const VkBindSparseInfo*                 pBindInfo,
    VkFence                                 fence);
```

- `queue` is the queue that the sparse binding operations will be submitted to.
- `bindInfoCount` is the number of elements in the pBindInfo array.
- `pBindInfo` is a pointer to an array of VkBindSparseInfo structures, each specifying a sparse binding submission batch.
- `fence` is an optional handle to a fence to be signaled. If `fence` is not VK_NULL_HANDLE, it defines a fence signal operation.

`vkQueueBindSparse` is a queue submission command, with each batch defined by an element of `pBindInfo` as a VkBindSparseInfo structure. Batches begin execution in the order they appear in `pBindInfo`, but may complete out of order.

Within a batch, a given range of a resource must not be bound more than once. Across batches, if a range is to be bound to one allocation and offset and then to another allocation and offset, then the application must guarantee (usually using semaphores) that the binding operations are executed in the correct order, as well as to order binding operations against the execution of command buffer submissions.

As no operation to `vkQueueBindSparse` causes any pipeline stage to access memory, synchronization primitives used in this command effectively only define execution dependencies.

Additional information about fence and semaphore operation is described in the synchronization chapter.
Valid Usage

- VUID-vkQueueBindSparse-fence-01113
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be unsignaled

- VUID-vkQueueBindSparse-fence-01114
  If `fence` is not `VK_NULL_HANDLE`, `fence` must not be associated with any other queue command that has not yet completed execution on that queue

- VUID-vkQueueBindSparse-pSignalSemaphores-01115
  Each element of the `pSignalSemaphores` member of each element of `pBindInfo` must be unsignaled when the semaphore signal operation it defines is executed on the device

- VUID-vkQueueBindSparse-pWaitSemaphores-01116
  When a semaphore wait operation referring to a binary semaphore defined by any element of the `pWaitSemaphores` member of any element of `pBindInfo` executes on `queue`, there must be no other queues waiting on the same semaphore

- VUID-vkQueueBindSparse-pWaitSemaphores-01117
  All elements of the `pWaitSemaphores` member of all elements of the `pBindInfo` parameter referring to a binary semaphore must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution

- VUID-vkQueueBindSparse-pWaitSemaphores-03245
  All elements of the `pWaitSemaphores` member of all elements of `pBindInfo` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY` must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution

Valid Usage (Implicit)

- VUID-vkQueueBindSparse-queue-parameter
  `queue` must be a valid `VkQueue` handle

- VUID-vkQueueBindSparse-pBindInfo-parameter
  If `bindInfoCount` is not 0, `pBindInfo` must be a valid pointer to an array of `bindInfoCount` valid `VkBindSparseInfo` structures

- VUID-vkQueueBindSparse-fence-parameter
  If `fence` is not `VK_NULL_HANDLE`, `fence` must be a valid `VkFence` handle

- VUID-vkQueueBindSparse-queuetype
  The `queue` must support sparse binding operations

- VUID-vkQueueBindSparse-commonparent
  Both of `fence`, and `queue` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`
Host Synchronization

- Host access to queue must be externally synchronized
- Host access to pBindInfo[].pBufferBinds[].buffer must be externally synchronized
- Host access to pBindInfo[].pImageOpaqueBinds[].image must be externally synchronized
- Host access to pBindInfo[].pImageBinds[].image must be externally synchronized
- Host access to fence must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>SPARSE_BINDING</td>
<td>-</td>
</tr>
</tbody>
</table>

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST

The VkBindSparseInfo structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkBindSparseInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    uint32_t bufferBindCount;
    const VkSparseBufferMemoryBindInfo* pBufferBinds;
    uint32_t imageOpaqueBindCount;
    const VkSparseImageOpaqueMemoryBindInfo* pImageOpaqueBinds;
    uint32_t imageBindCount;
    const VkSparseImageMemoryBindInfo* pImageBinds;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkBindSparseInfo;
```
• **sType** is the type of this structure.

• **pNext** is `NULL` or a pointer to a structure extending this structure.

• **waitSemaphoreCount** is the number of semaphores upon which to wait before executing the sparse binding operations for the batch.

• **pWaitSemaphores** is a pointer to an array of semaphores upon which to wait on before the sparse binding operations for this batch begin execution. If semaphores to wait on are provided, they define a **semaphore wait operation**.

• **bufferBindCount** is the number of sparse buffer bindings to perform in the batch.

• **pBufferBinds** is a pointer to an array of `VkSparseBufferMemoryBindInfo` structures.

• **imageOpaqueBindCount** is the number of opaque sparse image bindings to perform.

• **pImageOpaqueBinds** is a pointer to an array of `VkSparseImageOpaqueMemoryBindInfo` structures, indicating opaque sparse image bindings to perform.

• **imageBindCount** is the number of sparse image bindings to perform.

• **pImageBinds** is a pointer to an array of `VkSparseImageMemoryBindInfo` structures, indicating sparse image bindings to perform.

• **signalSemaphoreCount** is the number of semaphores to be signaled once the sparse binding operations specified by the structure have completed execution.

• **pSignalSemaphores** is a pointer to an array of semaphores which will be signaled when the sparse binding operations for this batch have completed execution. If semaphores to be signaled are provided, they define a **semaphore signal operation**.
Valid Usage

- VUID-VkBindSparseInfo-pWaitSemaphores-03246
  If any element of `pWaitSemaphores` or `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` then the `pNext` chain must include a `VkTimelineSemaphoreSubmitInfo` structure.

- VUID-VkBindSparseInfo-pNext-03247
  If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pWaitSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` then its `waitSemaphoreValueCount` member must equal `waitSemaphoreCount`.

- VUID-VkBindSparseInfo-pNext-03248
  If the `pNext` chain of this structure includes a `VkTimelineSemaphoreSubmitInfo` structure and any element of `pSignalSemaphores` was created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` then its `signalSemaphoreValueCount` member must equal `signalSemaphoreCount`.

- VUID-VkBindSparseInfo-pSignalSemaphores-03249
  For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value greater than the current value of the semaphore when the semaphore signal operation is executed.

- VUID-VkBindSparseInfo-pWaitSemaphores-03250
  For each element of `pWaitSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pWaitSemaphoreValues` must have a value which does not differ from the current value of the semaphore or from the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.

- VUID-VkBindSparseInfo-pSignalSemaphores-03251
  For each element of `pSignalSemaphores` created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_TIMELINE` the corresponding element of `VkTimelineSemaphoreSubmitInfo::pSignalSemaphoreValues` must have a value which does not differ from the current value of the semaphore or from the value of any outstanding semaphore wait or signal operation on that semaphore by more than `maxTimelineSemaphoreValueDifference`.
Valid Usage (Implicit)

- **VUID-VkBindSparseInfo-sType-sType**
  
  *sType* **must** be **VK_STRUCTURE_TYPE_BIND_SPARSE_INFO**

- **VUID-VkBindSparseInfo-pNext-pNext**
  
  Each *pNext* member of any structure (including this one) in the *pNext* chain **must** be either *NULL* or a pointer to a valid instance of **VkDeviceGroupBindSparseInfo** or **VkTimelineSemaphoreSubmitInfo**

- **VUID-VkBindSparseInfo-sType-unique**
  
  The *sType* value of each struct in the *pNext* chain **must** be unique

- **VUID-VkBindSparseInfo-pWaitSemaphores-parameter**
  
  If *waitSemaphoreCount* is not 0, *pWaitSemaphores** must** be a valid pointer to an array of *waitSemaphoreCount* valid **VkSemaphore** handles

- **VUID-VkBindSparseInfo-pBufferBinds-parameter**
  
  If *bufferBindCount* is not 0, *pBufferBinds** must** be a valid pointer to an array of *bufferBindCount* valid **VkSparseBufferMemoryBindInfo** structures

- **VUID-VkBindSparseInfo-pImageOpaqueBinds-parameter**
  
  If *imageOpaqueBindCount* is not 0, *pImageOpaqueBinds** must** be a valid pointer to an array of *imageOpaqueBindCount* valid **VkSparseImageOpaqueMemoryBindInfo** structures

- **VUID-VkBindSparseInfo-pImageBinds-parameter**
  
  If *imageBindCount* is not 0, *pImageBinds** must** be a valid pointer to an array of *imageBindCount* valid **VkSparseImageMemoryBindInfo** structures

- **VUID-VkBindSparseInfo-pSignalSemaphores-parameter**
  
  If *signalSemaphoreCount* is not 0, *pSignalSemaphores** must** be a valid pointer to an array of *signalSemaphoreCount* valid **VkSemaphore** handles

- **VUID-VkBindSparseInfo-commonparent**
  
  Both of the elements of *pSignalSemaphores*, and the elements of *pWaitSemaphores* that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same **VkDevice**

To specify the values to use when waiting for and signaling semaphores created with a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_TIMELINE**, add a **VkTimelineSemaphoreSubmitInfo** structure to the *pNext* chain of the **VkBindSparseInfo** structure.

If the *pNext* chain of **VkBindSparseInfo** includes a **VkDeviceGroupBindSparseInfo** structure, then that structure includes device indices specifying which instance of the resources and memory are bound.

The **VkDeviceGroupBindSparseInfo** structure is defined as:
typedef struct VkDeviceGroupBindSparseInfo {
    VkStructureType    sType;
    const void*        pNext;
    uint32_t           resourceDeviceIndex;
    uint32_t           memoryDeviceIndex;
} VkDeviceGroupBindSparseInfo;

or the equivalent

typedef VkDeviceGroupBindSparseInfo VkDeviceGroupBindSparseInfoKHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **resourceDeviceIndex** is a device index indicating which instance of the resource is bound.
- **memoryDeviceIndex** is a device index indicating which instance of the memory the resource instance is bound to.

These device indices apply to all buffer and image memory binds included in the batch pointing to this structure. The semaphore waits and signals for the batch are executed only by the physical device specified by the **resourceDeviceIndex**.

If this structure is not present, **resourceDeviceIndex** and **memoryDeviceIndex** are assumed to be zero.

### Valid Usage

- VUID-VkDeviceGroupBindSparseInfo-resourceDeviceIndex-01118
  
  **resourceDeviceIndex** and **memoryDeviceIndex** must both be valid device indices

- VUID-VkDeviceGroupBindSparseInfo-memoryDeviceIndex-01119
  
  Each memory allocation bound in this batch must have allocated an instance for **memoryDeviceIndex**

### Valid Usage (Implicit)

- VUID-VkDeviceGroupBindSparseInfo-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO**
Chapter 33. Window System Integration (WSI)

This chapter discusses the window system integration (WSI) between the Vulkan API and the various forms of displaying the results of rendering to a user. Since the Vulkan API can be used without displaying results, WSI is provided through the use of optional Vulkan extensions. This chapter provides an overview of WSI. See the appendix for additional details of each WSI extension, including which extensions must be enabled in order to use each of the functions described in this chapter.

33.1. WSI Platform

A platform is an abstraction for a window system, OS, etc. Some examples include MS Windows, Android, and Wayland. The Vulkan API may be integrated in a unique manner for each platform.

The Vulkan API does not define any type of platform object. Platform-specific WSI extensions are defined, each containing platform-specific functions for using WSI. Use of these extensions is guarded by preprocessor symbols as defined in the Window System-Specific Header Control appendix.

In order for an application to be compiled to use WSI with a given platform, it must either:

- #define the appropriate preprocessor symbol prior to including the vulkan.h header file, or
- include vulkan_core.h and any native platform headers, followed by the appropriate platform-specific header.

The preprocessor symbols and platform-specific headers are defined in the Window System Extensions and Headers table.

Each platform-specific extension is an instance extension. The application must enable instance extensions with vkCreateInstance before using them.

33.2. WSI Surface

Native platform surface or window objects are abstracted by surface objects, which are represented by VkSurfaceKHR handles:

```c
// Provided by VK_KHR_surface
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSurfaceKHR)
```

The VK_KHR_surface extension declares the VkSurfaceKHR object, and provides a function for destroying VkSurfaceKHR objects. Separate platform-specific extensions each provide a function for creating a VkSurfaceKHR object for the respective platform. From the application's perspective this is an opaque handle, just like the handles of other Vulkan objects.
On certain platforms, the Vulkan loader and ICDs may have conventions that treat the handle as a pointer to a structure containing the platform-specific information about the surface. This will be described in the documentation for the loader-ICD interface, and in the `vk_icd.h` header file of the LoaderAndTools source-code repository. This does not affect the loader-layer interface; layers may wrap `VkSurfaceKHR` objects.

### 33.2.1. Android Platform

To create a `VkSurfaceKHR` object for an Android native window, call:

```c
// Provided by VK_KHR_android_surface
VkResult vkCreateAndroidSurfaceKHR(
    VkInstance instance,
    const VkAndroidSurfaceCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkAndroidSurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

During the lifetime of a surface created using a particular `ANativeWindow` handle any attempts to create another surface for the same `ANativeWindow` and any attempts to connect to the same `ANativeWindow` through other platform mechanisms will fail.

In particular, only one `VkSurfaceKHR` can exist at a time for a given window. Similarly, a native window cannot be used by both a `VkSurfaceKHR` and `EGLSurface` simultaneously.

If successful, `vkCreateAndroidSurfaceKHR` increments the `ANativeWindow`'s reference count, and `vkDestroySurfaceKHR` will decrement it.

On Android, when a swapchain's `imageExtent` does not match the surface's `currentExtent`, the presentable images will be scaled to the surface's dimensions during presentation. `minImageExtent` is (1,1), and `maxImageExtent` is the maximum image size supported by the consumer. For the system compositor, `currentExtent` is the window size (i.e. the consumer's preferred size).
Valid Usage (Implicit)

- VUID-vkCreateAndroidSurfaceKHR-instance-parameter
  `instance` must be a valid `VkInstance` handle

- VUID-vkCreateAndroidSurfaceKHR-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid `VkAndroidSurfaceCreateInfoKHR` structure

- VUID-vkCreateAndroidSurfaceKHR-pAllocator-parameter
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateAndroidSurfaceKHR-pSurface-parameter
  `pSurface` must be a valid pointer to a `VkSurfaceKHR` handle

Return Codes

Success
- `VK_SUCCESS`

Failure
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkAndroidSurfaceCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_android_surface
typedef struct VkAndroidSurfaceCreateInfoKHR {
    VkStructureType                   sType;
    const void*                       pNext;
    VkAndroidSurfaceCreateFlagsKHR    flags;
    struct ANativeWindow*             window;
} VkAndroidSurfaceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `window` is a pointer to the `ANativeWindow` to associate the surface with.

Valid Usage

- VUID-VkAndroidSurfaceCreateInfoKHR-window-01248
  `window` must point to a valid Android `ANativeWindow`
Valid Usage (Implicit)

- **VUID-VkAndroidSurfaceCreateInfoKHR-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR*

- **VUID-VkAndroidSurfaceCreateInfoKHR-pNext-pNext**
  
  *pNext must be NULL*

- **VUID-VkAndroidSurfaceCreateInfoKHR-flags-zerobitmask**
  
  *flags must be 0*

To remove an unnecessary compile-time dependency, an incomplete type definition of `ANativeWindow` is provided in the Vulkan headers:

```c
// Provided by VK_KHR_android_surface
struct ANativeWindow;
```

The actual `ANativeWindow` type is defined in Android NDK headers.

```c
// Provided by VK_KHR_android_surface
typedef VkFlags VkAndroidSurfaceCreateFlagsKHR;
```

`VkAndroidSurfaceCreateFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.2. Wayland Platform

To create a `VkSurfaceKHR` object for a Wayland surface, call:

```c
// Provided by VK_KHR_wayland_surface
VkResult vkCreateWaylandSurfaceKHR(
    VkInstance                                  instance,
    const VkWaylandSurfaceCreateInfoKHR*        pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

- **instance** is the instance to associate the surface with.
- **pCreateInfo** is a pointer to a `VkWaylandSurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- **pSurface** is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.
Valid Usage (Implicit)

- VUID-vkCreateWaylandSurfaceKHR-instance-parameter
  instance must be a valid VkInstance handle

- VUID-vkCreateWaylandSurfaceKHR-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkWaylandSurfaceCreateInfoKHR structure

- VUID-vkCreateWaylandSurfaceKHR-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateWaylandSurfaceKHR-pSurface-parameter
  pSurface must be a valid pointer to a VkSurfaceKHR handle

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkWaylandSurfaceCreateInfoKHR structure is defined as:

```c
// Provided by VK_KHR_wayland_surface
typedef struct VkWaylandSurfaceCreateInfoKHR {
    VkStructureType                   sType;
    const void*                       pNext;
    VkWaylandSurfaceCreateFlagsKHR    flags;
    struct wl_display*                display;
    struct wl_surface*                surface;
} VkWaylandSurfaceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **display** and **surface** are pointers to the Wayland wl_display and wl_surface to associate the surface with.
On Wayland, \texttt{currentExtent} is the special value (0xFFFFFFFF, 0xFFFFFFFF), indicating that the surface size will be determined by the extent of a swapchain targeting the surface. Whatever the application sets a swapchain's \texttt{imageExtent} to will be the size of the window, after the first image is presented. \texttt{minImageExtent} is (1,1), and \texttt{maxImageExtent} is the maximum supported surface size. Any calls to \texttt{vkGetPhysicalDeviceSurfacePresentModesKHR} on a surface created with \texttt{vkCreateWaylandSurfaceKHR} are \textbf{required} to return \texttt{VK_PRESENT_MODE_MAILBOX_KHR} as one of the valid present modes.

Some Vulkan functions \textbf{may} send protocol over the specified \texttt{wl_display} connection when using a swapchain or presentable images created from a \texttt{VkSurfaceKHR} referring to a \texttt{wl_surface}. Applications \textbf{must} therefore ensure that both the \texttt{wl_display} and the \texttt{wl_surface} remain valid for the lifetime of any \texttt{VkSwapchainKHR} objects created from a particular \texttt{wl_display} and \texttt{wl_surface}. Also, calling \texttt{vkQueuePresentKHR} will result in Vulkan sending \texttt{wl_surface.commit} requests to the underlying \texttt{wl_surface} of each \texttt{VkSwapchainKHR} objects referenced by \texttt{pPresentInfo}. If the swapchain is created with a present mode of \texttt{VK_PRESENT_MODE_MAILBOX_KHR} or \texttt{VK_PRESENT_MODE_IMMEDIATE_KHR}, then the corresponding \texttt{wl_surface.attach}, \texttt{wl_surface.damage}, and \texttt{wl_surface.commit} request \textbf{must} be issued by the implementation during the call to \texttt{vkQueuePresentKHR} and \textbf{must} not be issued by the implementation outside of \texttt{vkQueuePresentKHR}. This ensures that any Wayland requests sent by the client after the call to \texttt{vkQueuePresentKHR} returns will be received by the compositor after the \texttt{wl_surface.commit}. Regardless of the mode of swapchain creation, a new \texttt{wl_event_queue} \textbf{must} be created for each successful \texttt{vkCreateWaylandSurfaceKHR} call, and every Wayland object created by the implementation \textbf{must} be assigned to this event queue. If the platform provides Wayland 1.11 or greater, this \textbf{must} be implemented by the use of Wayland proxy object wrappers, to avoid race conditions.

If the application wishes to synchronize any window changes with a particular frame, such requests \textbf{must} be sent to the Wayland display server prior to calling \texttt{vkQueuePresentKHR}. For full control over interactions between Vulkan rendering and other Wayland protocol requests and events, a present mode of \texttt{VK_PRESENT_MODE_MAILBOX_KHR} \textbf{should} be used.
VkWaylandSurfaceCreateFlagsKHR is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.3. Win32 Platform

To create a VkSurfaceKHR object for a Win32 window, call:

```c
// Provided by VK_KHR_win32_surface
VkResult vkCreateWin32SurfaceKHR(
    VkInstance instance,
    const VkWin32SurfaceCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);
```

- **instance** is the instance to associate the surface with.
- **pCreateInfo** is a pointer to a VkWin32SurfaceCreateInfoKHR structure containing parameters affecting the creation of the surface object.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- **pSurface** is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.

**Valid Usage (Implicit)**

- VUID-vkCreateWin32SurfaceKHR-instance-parameter
  
  **instance** must be a valid VkInstance handle

- VUID-vkCreateWin32SurfaceKHR-pCreateInfo-parameter
  
  **pCreateInfo** must be a valid pointer to a valid VkWin32SurfaceCreateInfoKHR structure

- VUID-vkCreateWin32SurfaceKHR-pAllocator-parameter
  
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateWin32SurfaceKHR-pSurface-parameter
  
  **pSurface** must be a valid pointer to a VkSurfaceKHR handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkWin32SurfaceCreateInfoKHR structure is defined as:

```c
// Provided by VK_KHR_win32_surface
typedef struct VkWin32SurfaceCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkWin32SurfaceCreateFlagsKHR flags;
    HINSTANCE hinstance;
    HWND hwnd;
} VkWin32SurfaceCreateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `hinstance` is the Win32 HINSTANCE for the window to associate the surface with.
- `hwnd` is the Win32 HWND for the window to associate the surface with.

Valid Usage
• VUID-VkWin32SurfaceCreateInfoKHR-hinstance-01307
  `hinstance` must be a valid Win32 HINSTANCE

• VUID-VkWin32SurfaceCreateInfoKHR-hwnd-01308
  `hwnd` must be a valid Win32 HWND

Valid Usage (Implicit)
• VUID-VkWin32SurfaceCreateInfoKHR-sType-sType
  `sType` must be VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR

• VUID-VkWin32SurfaceCreateInfoKHR-pNext-pNext
  `pNext` must be NULL

• VUID-VkWin32SurfaceCreateInfoKHR-flags-zerobitmask
  `flags` must be 0
With Win32, \texttt{minImageExtent}, \texttt{maxImageExtent}, and \texttt{currentExtent} \textbf{must} always equal the window size.

The \texttt{currentExtent} of a Win32 surface \textbf{must} have both \texttt{width} and \texttt{height} greater than 0, or both of them 0.

\begin{itemize}
  \item \texttt{currentExtent} of a Win32 surface \textbf{must} have both \texttt{width} and \texttt{height} greater than 0, or both of them 0.
\end{itemize}

\textbf{Note}

Due to above restrictions, it is only possible to create a new swapchain on this platform with \texttt{imageExtent} being equal to the current size of the window.

The window size \textbf{may} become (0, 0) on this platform (e.g. when the window is minimized), and so a swapchain \textbf{cannot} be created until the size changes.

\begin{verbatim}
// Provided by VK_KHR_win32_surface
typedef VkFlags VkWin32SurfaceCreateFlagsKHR;
\end{verbatim}

\texttt{VkWin32SurfaceCreateFlagsKHR} is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.4. XCB Platform

To create a \texttt{VkSurfaceKHR} object for an X11 window, using the XCB client-side library, call:

\begin{verbatim}
// Provided by VK_KHR_xcb_surface
VkResult vkCreateXcbSurfaceKHR(
    VkInstance                                  instance,
    const VkXcbSurfaceCreateInfoKHR*            pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
\end{verbatim}

- \texttt{instance} is the instance to associate the surface with.
- \texttt{pCreateInfo} is a pointer to a \texttt{VkXcbSurfaceCreateInfoKHR} structure containing parameters affecting the creation of the surface object.
- \texttt{pAllocator} is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- \texttt{pSurface} is a pointer to a \texttt{VkSurfaceKHR} handle in which the created surface object is returned.
Valid Usage (Implicit)

- **VUID-vkCreateXcbSurfaceKHR-instance-parameter**
  
  *instance* must be a valid [VkInstance](#) handle

- **VUID-vkCreateXcbSurfaceKHR-pCreateInfo-parameter**
  
  *pCreateInfo* must be a valid pointer to a valid [VkXcbSurfaceCreateInfoKHR](#) structure

- **VUID-vkCreateXcbSurfaceKHR-pAllocator-parameter**
  
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid [VkAllocationCallbacks](#) structure

- **VUID-vkCreateXcbSurfaceKHR-pSurface-parameter**
  
  *pSurface* must be a valid pointer to a [VkSurfaceKHR](#) handle

Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The [VkXcbSurfaceCreateInfoKHR](#) structure is defined as:

```c
// Provided by VK_KHR_xcb_surface
typedef struct VkXcbSurfaceCreateInfoKHR {
    VkStructureType               sType;
    const void                    *pNext;
    VkXcbSurfaceCreateFlagsKHR    flags;
    xcb_connection_t*             connection;
    xcb_window_t                  window;
} VkXcbSurfaceCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **connection** is a pointer to an *xcb_connection_t* to the X server.
- **window** is the *xcb_window_t* for the X11 window to associate the surface with.
Valid Usage

- VUID-VkXcbSurfaceCreateInfoKHR-connection-01310
  connection must point to a valid X11 xcb_connection_t

- VUID-VkXcbSurfaceCreateInfoKHR-window-01311
  window must be a valid X11 xcb_window_t

Valid Usage (Implicit)

- VUID-VkXcbSurfaceCreateInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR

- VUID-VkXcbSurfaceCreateInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkXcbSurfaceCreateInfoKHR-flags-zerobitmask
  flags must be 0

With Xcb, minImageExtent, maxImageExtent, and currentExtent must always equal the window size.

The currentExtent of an Xcb surface must have both width and height greater than 0, or both of them 0.

Note

Due to above restrictions, it is only possible to create a new swapchain on this platform with imageExtent being equal to the current size of the window.

The window size may become (0, 0) on this platform (e.g. when the window is minimized), and so a swapchain cannot be created until the size changes.

Some Vulkan functions may send protocol over the specified xcb connection when using a swapchain or presentable images created from a VkSurfaceKHR referring to an xcb window. Applications must therefore ensure the xcb connection is available to Vulkan for the duration of any functions that manipulate such swapchains or their presentable images, and any functions that build or queue command buffers that operate on such presentable images. Specifically, applications using Vulkan with xcb-based swapchains must

- Avoid holding a server grab on an xcb connection while waiting for Vulkan operations to complete using a swapchain derived from a different xcb connection referring to the same X server instance. Failing to do so may result in deadlock.

// Provided by VK_KHR_xcb_surface
typedef VkFlags VkXcbSurfaceCreateFlagsKHR;

VkXcbSurfaceCreateFlagsKHR is a bitmask type for setting a mask, but is currently reserved for future use.
33.2.5. Xlib Platform

To create a VkSurfaceKHR object for an X11 window, using the Xlib client-side library, call:

```c
// Provided by VK_KHR_xlib_surface
VkResult vkCreateXlibSurfaceKHR(
    VkInstance                                  instance,
    const VkXlibSurfaceCreateInfoKHR*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

- **instance** is the instance to associate the surface with.
- **pCreateInfo** is a pointer to a VkXlibSurfaceCreateInfoKHR structure containing the parameters affecting the creation of the surface object.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- **pSurface** is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.

**Valid Usage (Implicit)**

- VUID-vkCreateXlibSurfaceKHR-instance-parameter
  instance must be a valid VkInstance handle
- VUID-vkCreateXlibSurfaceKHR-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkXlibSurfaceCreateInfoKHR structure
- VUID-vkCreateXlibSurfaceKHR-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
- VUID-vkCreateXlibSurfaceKHR-pSurface-parameter
  pSurface must be a valid pointer to a VkSurfaceKHR handle

**Return Codes**

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkXlibSurfaceCreateInfoKHR structure is defined as:
typedef struct VkXlibSurfaceCreateInfoKHR {
    VkStructureType sType;
    const void*     pNext;
    VkXlibSurfaceCreateFlagsKHR flags;
    Display*        dpy;
    Window          window;
} VkXlibSurfaceCreateInfoKHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **dpy** is a pointer to an Xlib **Display** connection to the X server.
- **window** is an Xlib **Window** to associate the surface with.

### Valid Usage

- **VUID-VkXlibSurfaceCreateInfoKHR-dpy-01313**
  > dpy must point to a valid Xlib **Display**

- **VUID-VkXlibSurfaceCreateInfoKHR-window-01314**
  > window must be a valid Xlib **Window**

### Valid Usage (Implicit)

- **VUID-VkXlibSurfaceCreateInfoKHR-sType-sType**
  > sType must be **VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR**

- **VUID-VkXlibSurfaceCreateInfoKHR-pNext-pNext**
  > pNext must be **NULL**

- **VUID-VkXlibSurfaceCreateInfoKHR-flags-zerobitmask**
  > flags must be 0

With Xlib, **minImageExtent**, **maxImageExtent**, and **currentExtent** must always equal the window size.

The **currentExtent** of an Xlib surface must have both width and height greater than 0, or both of them 0.

**Note**

Due to above restrictions, it is only possible to create a new swapchain on this platform with **imageExtent** being equal to the current size of the window.

The window size may become (0, 0) on this platform (e.g. when the window is minimized), and so a swapchain cannot be created until the size changes.
Some Vulkan functions may send protocol over the specified Xlib Display connection when using a swapchain or presentable images created from a VkSurfaceKHR referring to an Xlib window. Applications must therefore ensure the display connection is available to Vulkan for the duration of any functions that manipulate such swapchains or their presentable images, and any functions that build or queue command buffers that operate on such presentable images. Specifically, applications using Vulkan with Xlib-based swapchains must

- Avoid holding a server grab on a display connection while waiting for Vulkan operations to complete using a swapchain derived from a different display connection referring to the same X server instance. Failing to do so may result in deadlock.

Some implementations may require threads to implement some presentation modes so applications must call XInitThreads() before calling any other Xlib functions.

```c
// Provided by VK_KHR_xlib_surface
typedef VkFlags VkXlibSurfaceCreateFlagsKHR;
```

VkXlibSurfaceCreateFlagsKHR is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.6. DirectFB Platform

To create a VkSurfaceKHR object for a DirectFB surface, call:

```c
// Provided by VK_EXT_directfb_surface
VkResult vkCreateDirectFBSurfaceEXT(
    VkInstance instance,
    const VkDirectFBSurfaceCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a VkDirectFBSurfaceCreateInfoEXT structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- `pSurface` is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.
Valid Usage (Implicit)

- VUID-vkCreateDirectFBSurfaceEXT-instance-parameter
  
  instance must be a valid VkInstance handle

- VUID-vkCreateDirectFBSurfaceEXT-pCreateInfo-parameter
  
  pCreateInfo must be a valid pointer to a valid VkDirectFBSurfaceCreateInfoEXT structure

- VUID-vkCreateDirectFBSurfaceEXT-pAllocator-parameter
  
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateDirectFBSurfaceEXT-pSurface-parameter
  
  pSurface must be a valid pointer to a VkSurfaceKHR handle

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDirectFBSurfaceCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_directfb_surface
typedef struct VkDirectFBSurfaceCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDirectFBSurfaceCreateFlagsEXT flags;
    IDirectFB* dfb;
    IDirectFBSurface* surface;
} VkDirectFBSurfaceCreateInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is reserved for future use.
- dfb is a pointer to the IDirectFB main interface of DirectFB.
- surface is a pointer to a IDirectFBSurface surface interface.
Valid Usage

- VUID-VkDirectFBSurfaceCreateInfoEXT-dfb-04117
  dfb must point to a valid DirectFB IDirectFB

- VUID-VkDirectFBSurfaceCreateInfoEXT-surface-04118
  surface must point to a valid DirectFB IDirectFBSurface

Valid Usage (Implicit)

- VUID-VkDirectFBSurfaceCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_DIRECTFB_SURFACE_CREATE_INFO_EXT

- VUID-VkDirectFBSurfaceCreateInfoEXT-pNext-pNext
  pNext must be NULL

- VUID-VkDirectFBSurfaceCreateInfoEXT-flags-zerobitmask
  flags must be 0

With DirectFB, minImageExtent, maxImageExtent, and currentExtent must always equal the surface size.

// Provided by VK_EXT_directfb_surface
typedef VkFlags VkDirectFBSurfaceCreateFlagsEXT;

VkDirectFBSurfaceCreateFlagsEXT is a bitmask type for setting a mask, but is currently reserved for future use.

33.2.7. Fuchsia Platform

To create a VkSurfaceKHR object for a Fuchsia ImagePipe, call:

// Provided by VK_FUCHSIA_imagepipe_surface
VkResult vkCreateImagePipeSurfaceFUCHSIA(
  VkInstance instance,
  const VkImagePipeSurfaceCreateInfoFUCHSIA* pCreateInfo,
  const VkAllocationCallbacks* pAllocator,
  VkSurfaceKHR* pSurface);

- instance is the instance to associate with the surface.
- pCreateInfo is a pointer to a VkImagePipeSurfaceCreateInfoFUCHSIA structure containing parameters affecting the creation of the surface object.
- pAllocator is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- pSurface is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.
Valid Usage (Implicit)

- VUID-vkCreateImagePipeSurfaceFUCHSIA-instance-parameter
  instance must be a valid VkInstance handle

- VUID-vkCreateImagePipeSurfaceFUCHSIA-pCreateInfo-parameter
  pCreateInfo must be a valid pointer to a valid VkImagePipeSurfaceCreateInfoFUCHSIA structure

- VUID-vkCreateImagePipeSurfaceFUCHSIA-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateImagePipeSurfaceFUCHSIA-pSurface-parameter
  pSurface must be a valid pointer to a VkSurfaceKHR handle

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkImagePipeSurfaceCreateInfoFUCHSIA structure is defined as:

```c
// Provided by VK_FUCHSIA_imagepipe_surface
typedef struct VkImagePipeSurfaceCreateInfoFUCHSIA {
    VkStructureType sType;
    const void* pNext;
    VkImagePipeSurfaceCreateFlagsFUCHSIA flags;
    zx_handle_t imagePipeHandle;
} VkImagePipeSurfaceCreateInfoFUCHSIA;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- flags is reserved for future use.
- imagePipeHandle is a zx_handle_t referring to the ImagePipe to associate with the surface.

Valid Usage

- VUID-VkImagePipeSurfaceCreateInfoFUCHSIA-imagePipeHandle-04863
  imagePipeHandle must be a valid zx_handle_t
Valid Usage (Implicit)

- VUID-VkImagePipeSurfaceCreateInfoFUCHSIA-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_IMAGEPIPE_SURFACE_CREATE_INFO_FUCHSIA

- VUID-VkImagePipeSurfaceCreateInfoFUCHSIA-pNext-pNext
  
  pNext must be NULL

- VUID-VkImagePipeSurfaceCreateInfoFUCHSIA-flags-zerobitmask
  
  flags must be 0

On Fuchsia, the surface currentExtent is the special value (0xFFFFFFFF, 0xFFFFFFFF), indicating that the surface size will be determined by the extent of a swapchain targeting the surface.

```c
// Provided by VK_FUCHSIA_imagepipe_surface
typedef VkFlags VkImagePipeSurfaceCreateFlagsFUCHSIA;
```

 VkImagePipeSurfaceCreateFlagsFUCHSIA is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.8. Google Games Platform

To create a VkSurfaceKHR object for a Google Games Platform stream descriptor, call:

```c
// Provided by VK_GGP_stream_descriptor_surface
VkResult vkCreateStreamDescriptorSurfaceGGP(
    VkInstance                                  instance,
    const VkStreamDescriptorSurfaceCreateInfoGGP* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

- instance is the instance to associate with the surface.
- pCreateInfo is a pointer to a VkStreamDescriptorSurfaceCreateInfoGGP structure containing parameters that affect the creation of the surface object.
- pAllocator is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- pSurface is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.
Valid Usage (Implicit)

- **VUID-vkCreateStreamDescriptorSurfaceGGP-instance-parameter**
  - *instance* must be a valid *VkInstance* handle

- **VUID-vkCreateStreamDescriptorSurfaceGGP-pCreateInfo-parameter**
  - *pCreateInfo* must be a valid pointer to a valid *VkStreamDescriptorSurfaceCreateInfoGGP* structure

- **VUID-vkCreateStreamDescriptorSurfaceGGP-pAllocator-parameter**
  - If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkCreateStreamDescriptorSurfaceGGP-pSurface-parameter**
  - *pSurface* must be a valid pointer to a *VkSurfaceKHR* handle

Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
- **VK_ERROR_NATIVE_WINDOW_IN_USE_KHR**

The *VkStreamDescriptorSurfaceCreateInfoGGP* structure is defined as:

```c
// Provided by VK_GGP_stream_descriptor_surface
typedef struct VkStreamDescriptorSurfaceCreateInfoGGP {
    VkStructureType             sType;
    const void*                 pNext;
    VkStreamDescriptorSurfaceCreateFlagsGGP flags;
    GgpStreamDescriptor         streamDescriptor;
} VkStreamDescriptorSurfaceCreateInfoGGP;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *flags* is reserved for future use.
- *streamDescriptor* is a *GgpStreamDescriptor* referring to the GGP stream descriptor to associate with the surface.
Valid Usage

- VUID-VkStreamDescriptorSurfaceCreateInfoGGP-streamDescriptor-02681
  streamDescriptor must be a valid GgpStreamDescriptor

Valid Usage (Implicit)

- VUID-VkStreamDescriptorSurfaceCreateInfoGGP-sType-sType
  sType must be VK_STRUCTURE_TYPE_STREAM_DESCRIPTOR_SURFACE_CREATE_INFO_GGP

- VUID-VkStreamDescriptorSurfaceCreateInfoGGP-pNext-pNext
  pNext must be NULL

- VUID-VkStreamDescriptorSurfaceCreateInfoGGP-flags-zerobitmask
  flags must be 0

On Google Games Platform, the surface extents are dynamic. The minImageExtent will never be greater than 1080p and the maxImageExtent will never be less than 1080p. The currentExtent will reflect the current optimal resolution.

Applications are expected to choose an appropriate size for the swapchain’s imageExtent, within the bounds of the surface. Using the surface’s currentExtent will offer the best performance and quality. When a swapchain's imageExtent does not match the surface's currentExtent, the presentable images are scaled to the surface’s dimensions during presentation if possible and VK_SUBOPTIMAL_KHR is returned, otherwise presentation fails with VK_ERROR_OUT_OF_DATE_KHR.

```cpp
// Provided by VK_GGP_stream_descriptor_surface
typedef VkFlags VkStreamDescriptorSurfaceCreateFlagsGGP;
```

VkStreamDescriptorSurfaceCreateFlagsGGP is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.9. iOS Platform

To create a VkSurfaceKHR object for an iOS UIView or CAMetalLayer, call:

```cpp
// Provided by VK_MVK_ios_surface
VkResult vkCreateIOSSurfaceMVK(
    VkInstance instance,
    const VkIOSSurfaceCreateInfoMVK* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);
```
Note

The `vkCreateIOSSurfaceMVK` function is considered deprecated and has been superseded by `vkCreateMetalSurfaceEXT` from the `VK_EXT_metal_surface` extension.

- **instance** is the instance with which to associate the surface.
- **pCreateInfo** is a pointer to a `VkIOSSurfaceCreateInfoMVK` structure containing parameters affecting the creation of the surface object.
- **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- **pSurface** is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateIOSSurfaceMVK-instance-parameter**  
  `instance` must be a valid `VkInstance` handle
- **VUID-vkCreateIOSSurfaceMVK-pCreateInfo-parameter**  
  `pCreateInfo` must be a valid pointer to a valid `VkIOSSurfaceCreateInfoMVK` structure
- **VUID-vkCreateIOSSurfaceMVK-pAllocator-parameter**  
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateIOSSurfaceMVK-pSurface-parameter**  
  `pSurface` must be a valid pointer to a `VkSurfaceKHR` handle

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
- **VK_ERROR_NATIVE_WINDOW_IN_USE_KHR**

The `VkIOSSurfaceCreateInfoMVK` structure is defined as:
typedef struct VkIOSSurfaceCreateInfoMVK {
    VkStructureType sType;
    const void* pNext;
    VkIOSSurfaceCreateFlagsMVK flags;
    const void* pView;
} VkIOSSurfaceCreateInfoMVK;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **pView** is a reference to either a CAMetalLayer object or a UIView object.

### Valid Usage

- VUID-VkIOSSurfaceCreateInfoMVK-pView-04143
  If **pView** is a CAMetalLayer object, it must be a valid CAMetalLayer.
- VUID-VkIOSSurfaceCreateInfoMVK-pView-01316
  If **pView** is a UIView object, it must be a valid UIView, must be backed by a CALayer object of type CAMetalLayer, and vkCreateIOSSurfaceMVK must be called on the main thread.

### Valid Usage (Implicit)

- VUID-VkIOSSurfaceCreateInfoMVK-sType-sType
  sType must be VK_STRUCTURE_TYPE_IOS_SURFACE_CREATE_INFO_MVK
- VUID-VkIOSSurfaceCreateInfoMVK-pNext-pNext
  pNext must be NULL
- VUID-VkIOSSurfaceCreateInfoMVK-flags-zerobitmask
  flags must be 0

// Provided by VK_MVK_ios_surface
typedef VkFlags VkIOSSurfaceCreateFlagsMVK;

VkIOSSurfaceCreateFlagsMVK is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.10. macOS Platform

To create a VkSurfaceKHR object for a macOS NSView or CAMetalLayer, call:
The `vkCreateMacOSSurfaceMVK` function is considered deprecated and has been superseded by `vkCreateMetalSurfaceEXT` from the `VK_EXT_metal_surface` extension.

- `instance` is the instance with which to associate the surface.
- `pCreateInfo` is a pointer to a `VkMacOSSurfaceCreateInfoMVK` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.

### Valid Usage (Implicit)

- `VUID-vkCreateMacOSSurfaceMVK-instance-parameter`  
  `instance` must be a valid `VkInstance` handle

- `VUID-vkCreateMacOSSurfaceMVK-pCreateInfo-parameter`  
  `pCreateInfo` must be a valid pointer to a valid `VkMacOSSurfaceCreateInfoMVK` structure

- `VUID-vkCreateMacOSSurfaceMVK-pAllocator-parameter`  
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- `VUID-vkCreateMacOSSurfaceMVK-pSurface-parameter`  
  `pSurface` must be a valid pointer to a `VkSurfaceKHR` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`

The `VkMacOSSurfaceCreateInfoMVK` structure is defined as:
typedef struct VkMacOSSurfaceCreateInfoMVK {
  VkStructureType     sType;
  const void*         pNext;
  VkMacOSSurfaceCreateFlagsMVK flags;
  const void*         pView;
} VkMacOSSurfaceCreateInfoMVK;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `pView` is a reference to either a `CAMetalLayer` object or an `NSView` object.

### Valid Usage

- VUID-VkMacOSSurfaceCreateInfoMVK-pView-04144
  If `pView` is a `CAMetalLayer` object, it must be a valid `CAMetalLayer`.

- VUID-VkMacOSSurfaceCreateInfoMVK-pView-01317
  If `pView` is an `NSView` object, it must be a valid `NSView`, must be backed by a `CALayer` object of type `CAMetalLayer`, and `vkCreateMacOSSurfaceMVK` must be called on the main thread.

### Valid Usage (Implicit)

- VUID-VkMacOSSurfaceCreateInfoMVK-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK`

- VUID-VkMacOSSurfaceCreateInfoMVK-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkMacOSSurfaceCreateInfoMVK-flags-zerobitmask
  `flags` must be `0`

typedef VkFlags VkMacOSSurfaceCreateFlagsMVK;

`VkMacOSSurfaceCreateFlagsMVK` is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.11. VI Platform

To create a `VkSurfaceKHR` object for an `nn::vi::Layer`, query the layer’s native handle using `nn::vi::GetNativeWindow`, and then call:
VkResult vkCreateViSurfaceNN(
    VkInstance                      instance,
    const VkViSurfaceCreateInfoNN*  pCreateInfo,
    const VkAllocationCallbacks*    pAllocator,
    VkSurfaceKHR*                   pSurface);

• **instance** is the instance with which to associate the surface.

• **pCreateInfo** is a pointer to a **VkViSurfaceCreateInfoNN** structure containing parameters affecting the creation of the surface object.

• **pAllocator** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).

• **pSurface** is a pointer to a **VkSurfaceKHR** handle in which the created surface object is returned.

During the lifetime of a surface created using a particular **nn::vi::NativeWindowHandle**, applications **must** not attempt to create another surface for the same **nn::vi::Layer** or attempt to connect to the same **nn::vi::Layer** through other platform mechanisms.

If the native window is created with a specified size, **currentExtent** will reflect that size. In this case, applications should use the same size for the swapchain’s **imageExtent**. Otherwise, the **currentExtent** will have the special value (0xFFFFFFFF, 0xFFFFFFFF), indicating that applications are expected to choose an appropriate size for the swapchain’s **imageExtent** (e.g., by matching the result of a call to **nn::vi::GetDisplayResolution**).

---

### Valid Usage (Implicit)

- **VUID-vkCreateViSurfaceNN-instance-parameter**
  
  **instance** **must** be a valid **VkInstance** handle

- **VUID-vkCreateViSurfaceNN-pCreateInfo-parameter**
  
  **pCreateInfo** **must** be a valid pointer to a valid **VkViSurfaceCreateInfoNN** structure

- **VUID-vkCreateViSurfaceNN-pAllocator-parameter**
  
  If **pAllocator** is not **NULL**, **pAllocator** **must** be a valid pointer to a valid **VkAllocationCallbacks** structure

- **VUID-vkCreateViSurfaceNN-pSurface-parameter**
  
  **pSurface** **must** be a valid pointer to a **VkSurfaceKHR** handle
Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_NATIVE_WINDOW_IN_USE_KHR

The `VkViSurfaceCreateInfoNN` structure is defined as:

```c
// Provided by VK_NN_vi_surface
typedef struct VkViSurfaceCreateInfoNN {
    VkStructureType sType;
    const void* pNext;
    VkViSurfaceCreateFlagsNN flags;
    void* window;
} VkViSurfaceCreateInfoNN;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `window` is the `nn::vi::NativeWindowHandle` for the `nn::vi::Layer` with which to associate the surface.

Valid Usage

- VUID-VkViSurfaceCreateInfoNN-window-01318
  `window` must be a valid `nn::vi::NativeWindowHandle`

Valid Usage (Implicit)

- VUID-VkViSurfaceCreateInfoNN-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN`

- VUID-VkViSurfaceCreateInfoNN-pNext-pNext
  `pNext` must be `NULL`

- VUID-VkViSurfaceCreateInfoNN-flags-zerobitmask
  `flags` must be 0


```c
// Provided by VK_NN_vi_surface
typedef VkFlags VkViSurfaceCreateFlagsNN;
```

**VkViSurfaceCreateFlagsNN** is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.12. Metal Platform

To create a *VkSurfaceKHR* object for a *CAMetalLayer*, call:

```c
// Provided by VK_EXT_metal_surface
VkResult vkCreateMetalSurfaceEXT(
    VkInstance instance,
    const VkMetalSurfaceCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkSurfaceKHR* pSurface);
```

- **`instance`** is the instance with which to associate the surface.
- **`pCreateInfo`** is a pointer to a *VkMetalSurfaceCreateInfoEXT* structure specifying parameters affecting the creation of the surface object.
- **`pAllocator`** is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see *Memory Allocation*).
- **`pSurface`** is a pointer to a *VkSurfaceKHR* handle in which the created surface object is returned.

### Valid Usage (Implicit)

- VUID-vkCreateMetalSurfaceEXT-instance-parameter
  - **`instance`** must be a valid *VkInstance* handle
- VUID-vkCreateMetalSurfaceEXT-pCreateInfo-parameter
  - **`pCreateInfo`** must be a valid pointer to a valid *VkMetalSurfaceCreateInfoEXT* structure
- VUID-vkCreateMetalSurfaceEXT-pAllocator-parameter
  - If **`pAllocator`** is not NULL, **`pAllocator`** must be a valid pointer to a valid *VkAllocationCallbacks* structure
- VUID-vkCreateMetalSurfaceEXT-pSurface-parameter
  - **`pSurface`** must be a valid pointer to a *VkSurfaceKHR* handle
Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_NATIVE_WINDOW_IN_USE_KHR

The `VkMetalSurfaceCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_metal_surface
typedef struct VkMetalSurfaceCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkMetalSurfaceCreateFlagsEXT flags;
    const CAMetalLayer* pLayer;
} VkMetalSurfaceCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `pLayer` is a reference to a `CAMetalLayer` object representing a renderable surface.

**Valid Usage (Implicit)**

- VUID-VkMetalSurfaceCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_METAL_SURFACE_CREATE_INFO_EXT`
- VUID-VkMetalSurfaceCreateInfoEXT-pNext-pNext
  `pNext` must be `NULL`
- VUID-VkMetalSurfaceCreateInfoEXT-flags-zerobitmask
  `flags` must be `0`

To remove an unnecessary compile-time dependency, an incomplete type definition of `CAMetalLayer` is provided in the Vulkan headers:
The actual `CAMetallayer` type is defined in the QuartzCore framework.

VkMetalSurfaceCreateFlagsEXT is a bitmask type for setting a mask, but is currently reserved for future use.

### 33.2.13. QNX Screen Platform

To create a `VkSurfaceKHR` object for a QNX Screen surface, call:

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a `VkScreenSurfaceCreateInfoQNX` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface object is returned.
Valid Usage (Implicit)

- **VUID-vkCreateScreenSurfaceQNX-instance-parameter**
  *instance* must be a valid *VkInstance* handle

- **VUID-vkCreateScreenSurfaceQNX-pCreateInfo-parameter**
  *pNextInfo* must be a valid pointer to a valid *VkScreenSurfaceCreateInfoQNX* structure

- **VUID-vkCreateScreenSurfaceQNX-pAllocator-parameter**
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkCreateScreenSurfaceQNX-pSurface-parameter**
  *pSurface* must be a valid pointer to a *VkSurfaceKHR* handle

Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

The *VkScreenSurfaceCreateInfoQNX* structure is defined as:

```c
// Provided by VK_QNX_screen_surface
typedef struct VkScreenSurfaceCreateInfoQNX {
    VkStructureType sType;
    const void* pNext;
    VkScreenSurfaceCreateFlagsQNX flags;
    struct _screen_context* context;
    struct _screen_window* window;
} VkScreenSurfaceCreateInfoQNX;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **context** and **window** are QNX Screen context and window to associate the surface with.
Valid Usage

• VUID-VkScreenSurfaceCreateInfoQNX-context-04741
  context must point to a valid QNX Screen struct _screen_context

• VUID-VkScreenSurfaceCreateInfoQNX-window-04742
  window must point to a valid QNX Screen struct _screen_window

Valid Usage (Implicit)

• VUID-VkScreenSurfaceCreateInfoQNX-sType-sType
  sType must be VK_STRUCTURE_TYPE_SCREEN_SURFACE_CREATE_INFO_QNX

• VUID-VkScreenSurfaceCreateInfoQNX-pNext-pNext
  pNext must be NULL

• VUID-VkScreenSurfaceCreateInfoQNX-flags-zerobitmask
  flags must be 0

// Provided by VK_QNX_screen_surface
typedef VkFlags VkScreenSurfaceCreateFlagsQNX;

VkScreenSurfaceCreateFlagsQNX is a bitmask type for setting a mask, but is currently reserved for future use.

33.2.14. Platform-Independent Information

Once created, VkSurfaceKHR objects can be used in this and other extensions, in particular the VK_KHR_swapchain extension.

Several WSI functions return VK_ERROR_SURFACE_LOST_KHR if the surface becomes no longer available. After such an error, the surface (and any child swapchain, if one exists) should be destroyed, as there is no way to restore them to a not-lost state. Applications may attempt to create a new VkSurfaceKHR using the same native platform window object, but whether such re-creation will succeed is platform-dependent and may depend on the reason the surface became unavailable. A lost surface does not otherwise cause devices to be lost.

To destroy a VkSurfaceKHR object, call:

// Provided by VK_KHR_surface
void vkDestroySurfaceKHR(
    VkInstance instance,
    VkSurfaceKHR surface,
    const VkAllocationCallbacks* pAllocator);

• instance is the instance used to create the surface.
• *surface* is the surface to destroy.

• *pAllocator* is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).

Destroying a *VkSurfaceKHR* merely severs the connection between Vulkan and the native surface, and does not imply destroying the native surface, closing a window, or similar behavior.

---

### Valid Usage

- VUID-vkDestroySurfaceKHR-surface-01266
  All *VkSwapchainKHR* objects created for *surface* must have been destroyed prior to destroying *surface*

- VUID-vkDestroySurfaceKHR-surface-01267
  If *VkAllocationCallbacks* were provided when *surface* was created, a compatible set of callbacks must be provided here

- VUID-vkDestroySurfaceKHR-surface-01268
  If no *VkAllocationCallbacks* were provided when *surface* was created, *pAllocator* must be NULL

### Valid Usage (Implicit)

- VUID-vkDestroySurfaceKHR-instance-parameter
  *instance* must be a valid *VkInstance* handle

- VUID-vkDestroySurfaceKHR-surface-parameter
  If *surface* is not *VK_NULL_HANDLE*, *surface* must be a valid *VkSurfaceKHR* handle

- VUID-vkDestroySurfaceKHR-pAllocator-parameter
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- VUID-vkDestroySurfaceKHR-surface-parent
  If *surface* is a valid handle, it must have been created, allocated, or retrieved from *instance*

### Host Synchronization

- Host access to *surface* must be externally synchronized

---

### 33.3. Presenting Directly to Display Devices

In some environments applications can also present Vulkan rendering directly to display devices without using an intermediate windowing system. This can be useful for embedded applications, or implementing the rendering/presentation backend of a windowing system using Vulkan. The *VK_KHR_display* extension provides the functionality necessary to enumerate display devices and create *VkSurfaceKHR* objects that target displays.
### 33.3.1. Display Enumeration

Displays are represented by `VkDisplayKHR` handles:

```c
// Provided by VK_KHR_display
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDisplayKHR)
```

Various functions are provided for enumerating the available display devices present on a Vulkan physical device. To query information about the available displays, call:

```c
// Provided by VK_KHR_display
VkResult vkGetPhysicalDeviceDisplayPropertiesKHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pPropertyCount,
    VkDisplayPropertiesKHR*                     pProperties);
```

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display devices available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayPropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display devices available for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display devices for `physicalDevice`, at most `pPropertyCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available properties were returned.

### Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceDisplayPropertiesKHR-physicalDevice-parameter`  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- `VUID-vkGetPhysicalDeviceDisplayPropertiesKHR-pPropertyCount-parameter`  
  `pPropertyCount` must be a valid pointer to a `uint32_t` value

- `VUID-vkGetPhysicalDeviceDisplayPropertiesKHR-pProperties-parameter`  
  If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkDisplayPropertiesKHR` structures
Return Codes

Success

• VK_SUCCESS
• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkDisplayPropertiesKHR` structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayPropertiesKHR {
    VkDisplayKHR display;
    const char* displayName;
    VkExtent2D physicalDimensions;
    VkExtent2D physicalResolution;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkBool32 planeReorderPossible;
    VkBool32 persistentContent;
} VkDisplayPropertiesKHR;
```

- `display` is a handle that is used to refer to the display described here. This handle will be valid for the lifetime of the Vulkan instance.

- `displayName` is a pointer to a null-terminated UTF-8 string containing the name of the display. Generally, this will be the name provided by the display’s EDID. It can be `NULL` if no suitable name is available. If not `NULL`, the memory it points to must remain accessible as long as `display` is valid.

- `physicalDimensions` describes the physical width and height of the visible portion of the display, in millimeters.

- `physicalResolution` describes the physical, native, or preferred resolution of the display.

- `supportedTransforms` is a bitmask of `VkSurfaceTransformFlagBitsKHR` describing which transforms are supported by this display.

- `planeReorderPossible` tells whether the planes on this display can have their z order changed. If this is `VK_TRUE`, the application can re-arrange the planes on this display in any order relative to each other.

- `persistentContent` tells whether the display supports self-refresh/internal buffering. If this is
true, the application can submit persistent present operations on swapchains created against this display.

**Note**

Persistent presents may have higher latency, and may use less power when the screen content is updated infrequently, or when only a portion of the screen needs to be updated in most frames.

To query information about the available displays, call:

```c
// Provided by VK_KHR_get_display_properties2
VkResult vkGetPhysicalDeviceDisplayProperties2KHR(
    VkPhysicalDevice physicalDevice,  // physicalDevice is a physical device.
    uint32_t* pPropertyCount,         // pPropertyCount is a pointer to an integer related to the number of display devices available or queried, as described below.
    VkDisplayProperties2KHR* pProperties);  // pProperties is either NULL or a pointer to an array of VkDisplayProperties2KHR structures.
```

vkGetPhysicalDeviceDisplayProperties2KHR behaves similarly to vkGetPhysicalDeviceDisplayPropertiesKHR, with the ability to return extended information via chained output structures.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceDisplayProperties2KHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceDisplayProperties2KHR-pPropertyCount-parameter
  pPropertyCount must be a valid pointer to a uint32_t value
- VUID-vkGetPhysicalDeviceDisplayProperties2KHR-pProperties-parameter
  If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties must be a valid pointer to an array of pPropertyCount VkDisplayProperties2KHR structures
**Return Codes**

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkDisplayProperties2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayProperties2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayPropertiesKHR displayProperties;
} VkDisplayProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `displayProperties` is a `VkDisplayPropertiesKHR` structure.

**Valid Usage (Implicit)**

- VUID-VkDisplayProperties2KHR-sType-sType
  `sType` **must** be `VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR`

- VUID-VkDisplayProperties2KHR-pNext-pNext
  `pNext` **must** be `NULL`

**Acquiring and Releasing Displays**

On some platforms, access to displays is limited to a single process or native driver instance. On such platforms, some or all of the displays may not be available to Vulkan if they are already in use by a native windowing system or other application.

To acquire permission to directly access a display in Vulkan from an X11 server, call:

```c
// Provided by VK_EXT_acquire_xlib_display
VkResult vkAcquireXlibDisplayEXT(
    VkPhysicalDevice   physicalDevice,
    Display*           dpy,
    VkDisplayKHR       display);
```
• **physicalDevice** The physical device the display is on.
• **dpy** A connection to the X11 server that currently owns **display**.
• **display** The display the caller wishes to control in Vulkan.

All permissions necessary to control the display are granted to the Vulkan instance associated with **physicalDevice** until the display is released or the X11 connection specified by **dpy** is terminated. Permission to access the display *may* be temporarily revoked during periods when the X11 server from which control was acquired itself loses access to **display**. During such periods, operations which require access to the display *must* fail with an appropriate error code. If the X11 server associated with **dpy** does not own **display**, or if permission to access it has already been acquired by another entity, the call *must* return the error code **VK_ERROR_INITIALIZATION_FAILED**.

**Note**
One example of when an X11 server loses access to a display is when it loses ownership of its virtual terminal.

### Valid Usage (Implicit)

- **VUID-vkAcquireXlibDisplayEXT-physicalDevice-parameter**
  - **physicalDevice** *must* be a valid **VkPhysicalDevice** handle
- **VUID-vkAcquireXlibDisplayEXT-dpy-parameter**
  - **dpy** *must* be a valid pointer to a **Display** value
- **VUID-vkAcquireXlibDisplayEXT-display-parameter**
  - **display** *must* be a valid **VkDisplayKHR** handle
- **VUID-vkAcquireXlibDisplayEXT-display-parent**
  - **display** *must* have been created, allocated, or retrieved from **physicalDevice**

### Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_INITIALIZATION_FAILED**

When acquiring displays from an X11 server, an application may also wish to enumerate and identify them using a native handle rather than a **VkDisplayKHR** handle. To determine the **VkDisplayKHR** handle corresponding to an X11 RandR Output, call:
// Provided by VK_EXT_acquire_xlib_display
VkResult vkGetRandROutputDisplayEXT(
    VkPhysicalDevice                            physicalDevice,
    Display*                                    dpy,
    RROutput                                    rrOutput,
    VkDisplayKHR*                               pDisplay);

• **physicalDevice** The physical device to query the display handle on.
• **dpy** A connection to the X11 server from which **rrOutput** was queried.
• **rrOutput** An X11 RandR output ID.
• **pDisplay** The corresponding **VkDisplayKHR** handle will be returned here.

If there is no **VkDisplayKHR** corresponding to **rrOutput** on **physicalDevice**, **VK_NULL_HANDLE** must be returned in **pDisplay**.

---

**Valid Usage (Implicit)**

• **VUID-vkGetRandROutputDisplayEXT-physicalDevice-parameter**
  physicalDevice **must** be a valid **VkPhysicalDevice** handle

• **VUID-vkGetRandROutputDisplayEXT-dpy-parameter**
  dpy **must** be a valid pointer to a **Display** value

• **VUID-vkGetRandROutputDisplayEXT-pDisplay-parameter**
  pDisplay **must** be a valid pointer to a **VkDisplayKHR** handle

---

**Return Codes**

**Success**

• **VK_SUCCESS**

**Failure**

• **VK_ERROR_OUT_OF_HOST_MEMORY**

---

To acquire permission to directly access a display in Vulkan on Windows 10, call:

// Provided by VK_NV_acquire_winrt_display
VkResult vkAcquireWinrtDisplayNV(
    VkPhysicalDevice                            physicalDevice,
    VkDisplayKHR                                display);

• **physicalDevice** The physical device the display is on.
• **display** The display the caller wishes to control in Vulkan.
All permissions necessary to control the display are granted to the Vulkan instance associated with `physicalDevice` until the display is released or the application is terminated. Permission to access the display may be revoked by events that cause Windows 10 itself to lose access to `display`. If this has happened, operations which require access to the display must fail with an appropriate error code. If permission to access `display` has already been acquired by another entity, the call must return the error code `VK_ERROR_INITIALIZATION_FAILED`.

### Note

The Vulkan instance acquires control of a “winrt::Windows::Devices::Display::Core::DisplayTarget” by performing an operation equivalent to “winrt::Windows::Devices::Display::Core::DisplayManager.TryAcquireTarget()” on the “DisplayTarget”.

### Note

One example of when Windows 10 loses access to a display is when the display is hot-unplugged.

### Note

One example of when a display has already been acquired by another entity is when the Windows desktop compositor (DWM) is in control of the display. Beginning with Windows 10 version 2004 it is possible to cause DWM to release a display by using the “Advanced display settings” sub-page of the “Display settings” control panel. `vkAcquireWinrtDisplayNV` does not itself cause DWM to release a display; this action must be performed outside of Vulkan.

### Valid Usage (Implicit)

- `VUID-vkAcquireWinrtDisplayNV-physicalDevice-parameter`
  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- `VUID-vkAcquireWinrtDisplayNV-display-parameter`
  
  `display` must be a valid `VkDisplayKHR` handle

- `VUID-vkAcquireWinrtDisplayNV-display-parent`
  
  `display` must have been created, allocated, or retrieved from `physicalDevice`
When acquiring displays on Windows 10, an application may also wish to enumerate and identify them using a native handle rather than a `VkDisplayKHR` handle.

To determine the `VkDisplayKHR` handle corresponding to a “winrt::Windows::Devices::Display::Core::DisplayTarget”, call:

```c
// Provided by VK_NV_acquire_winrt_display
VkResult vkGetWinrtDisplayNV(
    VkPhysicalDevice                            physicalDevice,
    uint32_t                                    deviceRelativeId,
    VkDisplayKHR*                               pDisplay);
```

- **physicalDevice** The physical device on which to query the display handle.
- **deviceRelativeId** The value of the “AdapterRelativeId” property of a “DisplayTarget” that is enumerated by a “DisplayAdapter” with an “Id” property matching the `deviceLUID` property of a `VkPhysicalDeviceIDProperties` for `physicalDevice`.
- **pDisplay** The corresponding `VkDisplayKHR` handle will be returned here.

If there is no `VkDisplayKHR` corresponding to `deviceRelativeId` on `physicalDevice`, `VK_NULL_HANDLE` must be returned in `pDisplay`.

### Valid Usage (Implicit)

- VUID-vkGetWinrtDisplayNV-physicalDevice-parameter  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetWinrtDisplayNV-pDisplay-parameter  
  `pDisplay` must be a valid pointer to a `VkDisplayKHR` handle
### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_INITIALIZATION_FAILED

To release a previously acquired display, call:

```c
// Provided by VK_EXT_direct_mode_display
VkResult vkReleaseDisplayEXT(
    VkPhysicalDevice                            physicalDevice,
    VkDisplayKHR                                display);
```

- **physicalDevice** The physical device the display is on.
- **display** The display to release control of.

### Valid Usage (Implicit)

- **VUID-vkReleaseDisplayEXT-physicalDevice-parameter**
  `physicalDevice` **must** be a valid `VkPhysicalDevice` handle

- **VUID-vkReleaseDisplayEXT-display-parameter**
  `display` **must** be a valid `VkDisplayKHR` handle

- **VUID-vkReleaseDisplayEXT-display-parent**
  `display` **must** have been created, allocated, or retrieved from `physicalDevice`

### Return Codes

**Success**
- VK_SUCCESS

**Display Planes**

Images are presented to individual planes on a display. Devices **must** support at least one plane on each display. Planes **can** be stacked and blended to composite multiple images on one display. Devices **may** support only a fixed stacking order and fixed mapping between planes and displays, or they **may** allow arbitrary application specified stacking orders and mappings between planes and displays. To query the properties of device display planes, call:
// Provided by VK_KHR_display
VkResult vkGetPhysicalDeviceDisplayPlanePropertiesKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayPlanePropertiesKHR* pProperties);

- `physicalDevice` is a physical device.
- `pPropertyCount` is a pointer to an integer related to the number of display planes available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayPlanePropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display planes available for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display planes for `physicalDevice`, at most `pPropertyCount` structures will be written.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceDisplayPlanePropertiesKHR-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceDisplayPlanePropertiesKHR-pPropertyCount-parameter `pPropertyCount` must be a valid pointer to a `uint32_t` value
- VUID-vkGetPhysicalDeviceDisplayPlanePropertiesKHR-pProperties-parameter If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkDisplayPlanePropertiesKHR` structures

**Return Codes**

**Success**

- `VK_SUCCESS`
- `VK_INCOMPLETE`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlanePropertiesKHR` structure is defined as:

---

Chapter 33. Window System Integration (WSI) | 1751
// Provided by VK_KHR_display
typedef struct VkDisplayPlanePropertiesKHR {
    VkDisplayKHR currentDisplay;
    uint32_t currentStackIndex;
} VkDisplayPlanePropertiesKHR;

- **currentDisplay** is the handle of the display the plane is currently associated with. If the plane is not currently attached to any displays, this will be **VK_NULL_HANDLE**.

- **currentStackIndex** is the current z-order of the plane. This will be between 0 and the value returned by `vkGetPhysicalDeviceDisplayPlanePropertiesKHR` in `pPropertyCount`.

To query the properties of a device’s display planes, call:

// Provided by VK_KHR_get_display_properties2
VkResult vkGetPhysicalDeviceDisplayPlaneProperties2KHR(
    VkPhysicalDevice physicalDevice,
    uint32_t* pPropertyCount,
    VkDisplayPlaneProperties2KHR* pProperties);

- **physicalDevice** is a physical device.

- **pPropertyCount** is a pointer to an integer related to the number of display planes available or queried, as described below.

- **pProperties** is either **NULL** or a pointer to an array of `VkDisplayPlaneProperties2KHR` structures.

`vkGetPhysicalDeviceDisplayPlaneProperties2KHR` behaves similarly to `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`, with the ability to return extended information via chained output structures.

### Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceDisplayPlaneProperties2KHR-physicalDevice-parameter**
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- **VUID-vkGetPhysicalDeviceDisplayPlaneProperties2KHR-pPropertyCount-parameter**
  `pPropertyCount` must be a valid pointer to a `uint32_t` value

- **VUID-vkGetPhysicalDeviceDisplayPlaneProperties2KHR-pProperties-parameter**
  If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not **NULL**, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkDisplayPlaneProperties2KHR` structures
Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkDisplayPlaneProperties2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayPlaneProperties2KHR {
    VkStructureType sType;                  // Provided by VK_KHR_get_display_properties2
    void* pNext;                            // Provided by VK_KHR_get_display_properties2
    VkDisplayPlanePropertiesKHR displayPlaneProperties; // Provided by VK_KHR_get_display_properties2
} VkDisplayPlaneProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `displayPlaneProperties` is a `VkDisplayPlanePropertiesKHR` structure.

Valid Usage (Implicit)

- VUID-VkDisplayPlaneProperties2KHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR`
- VUID-VkDisplayPlaneProperties2KHR-pNext-pNext
  `pNext` must be `NULL`

To determine which displays a plane is usable with, call

```c
// Provided by VK_KHR_display
VkResult vkGetDisplayPlaneSupportedDisplaysKHR(
    VkPhysicalDevice physicalDevice,              // Provided by VK_KHR_display
    uint32_t planeIndex,                         // Provided by VK_KHR_display
    uint32_t* pDisplayCount,                     // Provided by VK_KHR_display
    VkDisplayKHR* pDisplays);                   // Provided by VK_KHR_display
```

- `physicalDevice` is a physical device.
- `planeIndex` is the plane which the application wishes to use, and must be in the range [0, physical device plane count - 1].
• `pDisplayCount` is a pointer to an integer related to the number of displays available or queried, as described below.

• `pDisplays` is either `NULL` or a pointer to an array of `VkDisplayKHR` handles.

If `pDisplays` is `NULL`, then the number of displays usable with the specified `planeIndex` for `physicalDevice` is returned in `pDisplayCount`. Otherwise, `pDisplayCount` must point to a variable set by the user to the number of elements in the `pDisplays` array, and on return the variable is overwritten with the number of handles actually written to `pDisplays`. If the value of `pDisplayCount` is less than the number of usable display-plane pairs for `physicalDevice`, at most `pDisplayCount` handles will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available pairs were returned.

### Valid Usage

- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-planeIndex-01249
  `planeIndex` must be less than the number of display planes supported by the device as determined by calling `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`

### Valid Usage (Implicit)

- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-pDisplayCount-parameter
  `pDisplayCount` must be a valid pointer to a `uint32_t` value

- VUID-vkGetDisplayPlaneSupportedDisplaysKHR-pDisplays-parameter
  If the value referenced by `pDisplayCount` is not 0, and `pDisplays` is not `NULL`, `pDisplays` must be a valid pointer to an array of `pDisplayCount` `VkDisplayKHR` handles

### Return Codes

#### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

Additional properties of displays are queried using specialized query functions.

**Display Modes**

Display modes are represented by `VkDisplayModeKHR` handles:
Each display has one or more supported modes associated with it by default. These built-in modes are queried by calling:

```c
// Provided by VK_KHR_display
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDisplayModeKHR)

VkResult vkGetDisplayModePropertiesKHR(
    VkPhysicalDevice                            physicalDevice,
    VkDisplayKHR                                display,
    uint32_t*                                   pPropertyCount,
    VkDisplayModePropertiesKHR*                 pProperties);
```

- `physicalDevice` is the physical device associated with `display`.
- `display` is the display to query.
- `pPropertyCount` is a pointer to an integer related to the number of display modes available or queried, as described below.
- `pProperties` is either `NULL` or a pointer to an array of `VkDisplayModePropertiesKHR` structures.

If `pProperties` is `NULL`, then the number of display modes available on the specified `display` for `physicalDevice` is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If the value of `pPropertyCount` is less than the number of display modes for `physicalDevice`, at most `pPropertyCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available display modes were returned.

### Valid Usage (Implicit)

- VUID-vkGetDisplayModePropertiesKHR-physicalDevice-parameter
  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetDisplayModePropertiesKHR-display-parameter
  
  `display` must be a valid `VkDisplayKHR` handle

- VUID-vkGetDisplayModePropertiesKHR-pPropertyCount-parameter
  
  `pPropertyCount` must be a valid pointer to a `uint32_t` value

- VUID-vkGetDisplayModePropertiesKHR-pProperties-parameter
  
  If the value referenced by `pPropertyCount` is not `0`, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkDisplayModePropertiesKHR` structures

- VUID-vkGetDisplayModePropertiesKHR-display-parent
  
  `display` must have been created, allocated, or retrieved from `physicalDevice`
Return Codes

Success
• VK_SUCCESS
• VK_INCOMPLETE

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDisplayModePropertiesKHR structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayModePropertiesKHR {
    VkDisplayModeKHR              displayMode;
    VkDisplayModeParametersKHR    parameters;
} VkDisplayModePropertiesKHR;
```

- `displayMode` is a handle to the display mode described in this structure. This handle will be valid for the lifetime of the Vulkan instance.
- `parameters` is a `VkDisplayModeParametersKHR` structure describing the display parameters associated with `displayMode`.

```c
// Provided by VK_KHR_display
typedef VkFlags VkDisplayModeCreateFlagsKHR;
```

`VkDisplayModeCreateFlagsKHR` is a bitmask type for setting a mask, but is currently reserved for future use.

To query the properties of a device’s built-in display modes, call:

```c
// Provided by VK_KHR_get_display_properties2
VkResult vkGetDisplayModeProperties2KHR(
    VkPhysicalDevice                            physicalDevice,
    VkDisplayKHR                                display,
    uint32_t*                                   pPropertyCount,
    VkDisplayModeProperties2KHR*                pProperties);
```

- `physicalDevice` is the physical device associated with `display`.
- `display` is the display to query.
- `pPropertyCount` is a pointer to an integer related to the number of display modes available or queried, as described below.
• pProperties is either NULL or a pointer to an array of VkDisplayModeProperties2KHR structures.

vkGetDisplayModeProperties2KHR behaves similarly to vkGetDisplayModePropertiesKHR, with the ability to return extended information via chained output structures.

Valid Usage (Implicit)

• VUID-vkGetDisplayModeProperties2KHR-physicalDevice-parameter 
  physicalDevice must be a valid VkPhysicalDevice handle

• VUID-vkGetDisplayModeProperties2KHR-display-parameter
  display must be a valid VkDisplayKHR handle

• VUID-vkGetDisplayModeProperties2KHR-pPropertyCount-parameter
  pPropertyCount must be a valid pointer to a uint32_t value

• VUID-vkGetDisplayModeProperties2KHR-pProperties-parameter
  If the value referenced by pPropertyCount is not 0, and pProperties is not NULL, pProperties must be a valid pointer to an array of pPropertyCount VkDisplayModeProperties2KHR structures

• VUID-vkGetDisplayModeProperties2KHR-display-parent
  display must have been created, allocated, or retrieved from physicalDevice

Return Codes

Success

• VK_SUCCESS
• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDisplayModeProperties2KHR structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayModeProperties2KHR {
  VkStructureType   sType;
  void*             pNext;
  VkDisplayModePropertiesKHR displayModeProperties;
} VkDisplayModeProperties2KHR;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• displayModeProperties is a VkDisplayModePropertiesKHR structure.
Valid Usage (Implicit)

- VUID-VkDisplayModeProperties2KHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR

- VUID-VkDisplayModeProperties2KHR-pNext-pNext
  pNext must be NULL

The VkDisplayModeParametersKHR structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayModeParametersKHR {
    VkExtent2D    visibleRegion;
    uint32_t      refreshRate;
} VkDisplayModeParametersKHR;
```

- **visibleRegion** is the 2D extents of the visible region.
- **refreshRate** is a `uint32_t` that is the number of times the display is refreshed each second multiplied by 1000.

**Note**
For example, a 60Hz display mode would report a `refreshRate` of 60,000.

Valid Usage

- VUID-VkDisplayModeParametersKHR-width-01990
  The width member of `visibleRegion` must be greater than 0

- VUID-VkDisplayModeParametersKHR-height-01991
  The height member of `visibleRegion` must be greater than 0

- VUID-VkDisplayModeParametersKHR-refreshRate-01992
  `refreshRate` must be greater than 0

Additional modes may also be created by calling:

```c
// Provided by VK_KHR_display
VkResult vkCreateDisplayModeKHR(  
    VkPhysicalDevice                            physicalDevice,  
    VkDisplayKHR                                display,  
    const VkDisplayModeCreateInfoKHR*           pCreateInfo,  
    const VkAllocationCallbacks*                pAllocator,  
    VkDisplayModeKHR*                           pMode);
```

- **physicalDevice** is the physical device associated with **display**.
• **display** is the display to create an additional mode for.

• **pCreateInfo** is a `VkDisplayModeCreateInfoKHR` structure describing the new mode to create.

• **pAllocator** is the allocator used for host memory allocated for the display mode object when there is no more specific allocator available (see Memory Allocation).

• **pMode** returns the handle of the mode created.

### Valid Usage (Implicit)

- VUID-vkCreateDisplayModeKHR-physicalDevice-parameter  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkCreateDisplayModeKHR-display-parameter  
  `display` must be a valid `VkDisplayKHR` handle

- VUID-vkCreateDisplayModeKHR-pCreateInfo-parameter  
  `pCreateInfo` must be a valid pointer to a valid `VkDisplayModeCreateInfoKHR` structure

- VUID-vkCreateDisplayModeKHR-pAllocator-parameter  
  If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateDisplayModeKHR-pMode-parameter  
  `pMode` must be a valid pointer to a `VkDisplayModeKHR` handle

- VUID-vkCreateDisplayModeKHR-display-parent  
  `display` must have been created, allocated, or retrieved from `physicalDevice`

### Host Synchronization

- Host access to `display` must be externally synchronized

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED

The `VkDisplayModeCreateInfoKHR` structure is defined as:
// Provided by VK_KHR_display
typedef struct VkDisplayModeCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDisplayModeCreateFlagsKHR flags;
    VkDisplayModeParametersKHR parameters;
} VkDisplayModeCreateInfoKHR;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **flags** is reserved for future use, and must be zero.
• **parameters** is a VkDisplayModeParametersKHR structure describing the display parameters to use in creating the new mode. If the parameters are not compatible with the specified display, the implementation must return **VK_ERROR_INITIALIZATION_FAILED**.

### Valid Usage (Implicit)

- **VUID-VkDisplayModeCreateInfoKHR-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR`
- **VUID-VkDisplayModeCreateInfoKHR-pNext-pNext**
  `pNext` must be **NULL**
- **VUID-VkDisplayModeCreateInfoKHR-flags-zerobitmask**
  `flags` must be 0
- **VUID-VkDisplayModeCreateInfoKHR-parameters-parameter**
  `parameters` must be a valid `VkDisplayModeParametersKHR` structure

Applications that wish to present directly to a display must select which layer, or “plane” of the display they wish to target, and a mode to use with the display. Each display supports at least one plane. The capabilities of a given mode and plane combination are determined by calling:

// Provided by VK_KHR_display
VkResult vkGetDisplayPlaneCapabilitiesKHR(
    VkPhysicalDevice physicalDevice,
    VkDisplayModeKHR mode,
    uint32_t planeIndex,
    VkDisplayPlaneCapabilitiesKHR* pCapabilities);

• **physicalDevice** is the physical device associated with the display specified by **mode**
• **mode** is the display mode the application intends to program when using the specified plane. Note this parameter also implicitly specifies a display.
• **planeIndex** is the plane which the application intends to use with the display, and is less than the number of display planes supported by the device.
• **pCapabilities** is a pointer to a `VkDisplayPlaneCapabilitiesKHR` structure in which the capabilities are returned.

### Valid Usage (Implicit)

- **VUID-vkGetDisplayPlaneCapabilitiesKHR-physicalDevice-parameter**
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle.
- **VUID-vkGetDisplayPlaneCapabilitiesKHR-mode-parameter**
  - `mode` must be a valid `VkDisplayModeKHR` handle.
- **VUID-vkGetDisplayPlaneCapabilitiesKHR-pCapabilities-parameter**
  - `pCapabilities` must be a valid pointer to a `VkDisplayPlaneCapabilitiesKHR` structure.

### Host Synchronization

- Host access to `mode` must be externally synchronized.

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplayPlaneCapabilitiesKHR` structure is defined as:

```c
// Provided by VK_KHR_display
typedef struct VkDisplayPlaneCapabilitiesKHR {
    VkDisplayPlaneAlphaFlagsKHR    supportedAlpha;
    VkOffset2D                     minSrcPosition;
    VkOffset2D                     maxSrcPosition;
    VkExtent2D                     minSrcExtent;
    VkExtent2D                     maxSrcExtent;
    VkOffset2D                     minDstPosition;
    VkOffset2D                     maxDstPosition;
    VkExtent2D                     minDstExtent;
    VkExtent2D                     maxDstExtent;
} VkDisplayPlaneCapabilitiesKHR;
```

- **supportedAlpha** is a bitmask of `VkDisplayPlaneAlphaFlagBitsKHR` describing the supported alpha blending modes.
- **minSrcPosition** is the minimum source rectangle offset supported by this plane using the
specified mode.

- **maxSrcPosition** is the maximum source rectangle offset supported by this plane using the specified mode. The x and y components of **maxSrcPosition** must each be greater than or equal to the x and y components of **minSrcPosition**, respectively.

- **minSrcExtent** is the minimum source rectangle size supported by this plane using the specified mode.

- **maxSrcExtent** is the maximum source rectangle size supported by this plane using the specified mode.

- **minDstPosition**, **maxDstPosition**, **minDstExtent**, **maxDstExtent** all have similar semantics to their corresponding *Src* equivalents, but apply to the output region within the mode rather than the input region within the source image. Unlike the *Src* offsets, **minDstPosition** and **maxDstPosition** may contain negative values.

The minimum and maximum position and extent fields describe the implementation limits, if any, as they apply to the specified display mode and plane. Vendors may support displaying a subset of a swapchain's presentable images on the specified display plane. This is expressed by returning **minSrcPosition**, **maxSrcPosition**, **minSrcExtent**, and **maxSrcExtent** values that indicate a range of possible positions and sizes which may be used to specify the region within the presentable images that source pixels will be read from when creating a swapchain on the specified display mode and plane.

Vendors may also support mapping the presentable images' content to a subset or superset of the visible region in the specified display mode. This is expressed by returning **minDstPosition**, **maxDstPosition**, **minDstExtent**, and **maxDstExtent** values that indicate a range of possible positions and sizes which may be used to describe the region within the display mode that the source pixels will be mapped to.

Other vendors may support only a 1-1 mapping between pixels in the presentable images and the display mode. This may be indicated by returning (0,0) for **minSrcPosition**, **maxSrcPosition**, **minDstPosition**, and **maxDstPosition**, and (display mode width, display mode height) for **minSrcExtent**, **maxSrcExtent**, **minDstExtent**, and **maxDstExtent**.

The value supportedAlpha must contain at least one valid VkDisplayPlaneAlphaFlagBitsKHR bit.

These values indicate the limits of the implementation's individual fields. Not all combinations of values within the offset and extent ranges returned in VkDisplayPlaneCapabilitiesKHR are guaranteed to be supported. Presentation requests specifying unsupported combinations may fail.

To query the capabilities of a given mode and plane combination, call:

```c
// Provided by VK_KHR_get_display_properties2
VkResult vkGetDisplayPlaneCapabilities2KHR(
    VkPhysicalDevice physicalDevice,
    const VkDisplayPlaneInfo2KHR* pDisplayPlaneInfo,
    VkDisplayPlaneCapabilities2KHR* pCapabilities);
```

- **physicalDevice** is the physical device associated with **pDisplayPlaneInfo**.
• **pDisplayPlaneInfo** is a pointer to a *VkDisplayPlaneInfo2KHR* structure describing the plane and mode.

• **pCapabilities** is a pointer to a *VkDisplayPlaneCapabilities2KHR* structure in which the capabilities are returned.

`vkGetDisplayPlaneCapabilities2KHR` behaves similarly to `vkGetDisplayPlaneCapabilitiesKHR`, with the ability to specify extended inputs via chained input structures, and to return extended information via chained output structures.

---

### Valid Usage (Implicit)

- **VUID-vkGetDisplayPlaneCapabilities2KHR-physicalDevice-parameter**
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle

- **VUID-vkGetDisplayPlaneCapabilities2KHR-pDisplayPlaneInfo-parameter**
  - `pDisplayPlaneInfo` must be a valid pointer to a valid `VkDisplayPlaneInfo2KHR` structure

- **VUID-vkGetDisplayPlaneCapabilities2KHR-pCapabilities-parameter**
  - `pCapabilities` must be a valid pointer to a `VkDisplayPlaneCapabilities2KHR` structure

---

### Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

---

The `VkDisplayPlaneInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayPlaneInfo2KHR {
    VkStructureType     sType;
    const void*         pNext;
    VkDisplayModeKHR    mode;
    uint32_t            planeIndex;
} VkDisplayPlaneInfo2KHR;
```

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **mode** is the display mode the application intends to program when using the specified plane.
### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR`
- **pNext** must be `NULL`
- `planeIndex` is the plane which the application intends to use with the display.

### Host Synchronization

- Host access to **mode** must be externally synchronized

The **VkDisplayPlaneCapabilities2KHR** structure is defined as:

```c
// Provided by VK_KHR_get_display_properties2
typedef struct VkDisplayPlaneCapabilities2KHR {
    VkStructureType sType;
    void* pNext;
    VkDisplayPlaneCapabilitiesKHR capabilities;
} VkDisplayPlaneCapabilities2KHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **capabilities** is a **VkDisplayPlaneCapabilitiesKHR** structure.

### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR`
- **pNext** must be `NULL`
33.3.2. Display Control

To set the power state of a display, call:

```c
// Provided by VK_EXT_display_control
VkResult vkDisplayPowerControlEXT(
    VkDevice                                    device,
    VkDisplayKHR                                display,
    const VkDisplayPowerInfoEXT*                pDisplayPowerInfo);
```

- `device` is a logical device associated with `display`.
- `display` is the display whose power state is modified.
- `pDisplayPowerInfo` is a `VkDisplayPowerInfoEXT` structure specifying the new power state of `display`.

**Valid Usage (Implicit)**

- VUID-vkDisplayPowerControlEXT-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkDisplayPowerControlEXT-display-parameter
  - `display` must be a valid `VkDisplayKHR` handle
- VUID-vkDisplayPowerControlEXT-pDisplayPowerInfo-parameter
  - `pDisplayPowerInfo` must be a valid pointer to a valid `VkDisplayPowerInfoEXT` structure
- VUID-vkDisplayPowerControlEXT-commonparent
  - Both of `device`, and `display` must have been created, allocated, or retrieved from the same `VkPhysicalDevice`

**Return Codes**

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`

The `VkDisplayPowerInfoEXT` structure is defined as:
// Provided by VK_EXT_display_control
typedef struct VkDisplayPowerInfoEXT {
    VkStructureType           sType;
    const void*               pNext;
    VkDisplayPowerStateEXT    powerState;
} VkDisplayPowerInfoEXT;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **powerState** is a **VkDisplayPowerStateEXT** value specifying the new power state of the display.

### Valid Usage (Implicit)

- **VUID-VkDisplayPowerInfoEXT-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT**

- **VUID-VkDisplayPowerInfoEXT-pNext-pNext**
  
  **pNext** must be **NULL**

- **VUID-VkDisplayPowerInfoEXT-powerState-parameter**
  
  **powerState** must be a valid **VkDisplayPowerStateEXT** value

Possible values of **VkDisplayPowerInfoEXT::powerState**, specifying the new power state of a display, are:

// Provided by VK_EXT_display_control
typedef enum VkDisplayPowerStateEXT {
    VK_DISPLAY_POWER_STATE_OFF_EXT = 0,
    VK_DISPLAY_POWER_STATE_SUSPEND_EXT = 1,
    VK_DISPLAY_POWER_STATE_ON_EXT = 2,
} VkDisplayPowerStateEXT;

• **VK_DISPLAY_POWER_STATE_OFF_EXT** specifies that the display is powered down.
• **VK_DISPLAY_POWER_STATE_SUSPEND_EXT** specifies that the display is put into a low power mode, from which it **may** be able to transition back to **VK_DISPLAY_POWER_STATE_ON_EXT** more quickly than if it were in **VK_DISPLAY_POWER_STATE_OFF_EXT**. This state **may** be the same as **VK_DISPLAY_POWER_STATE_OFF_EXT**.
• **VK_DISPLAY_POWER_STATE_ON_EXT** specifies that the display is powered on.

### 33.3.3. Display Surfaces

A complete display configuration includes a mode, one or more display planes and any parameters describing their behavior, and parameters describing some aspects of the images associated with those planes. Display surfaces describe the configuration of a single plane within a complete display configuration. To create a **VkSurfaceKHR** object for a display plane, call:
Provided by VK_KHR_display

```c
VkResult vkCreateDisplayPlaneSurfaceKHR(
    VkInstance                                  instance,
    const VkDisplaySurfaceCreateInfoKHR*        pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

- `instance` is the instance corresponding to the physical device the targeted display is on.
- `pCreateInfo` is a pointer to a `VkDisplaySurfaceCreateInfoKHR` structure specifying which mode, plane, and other parameters to use, as described below.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- `pSurface` is a pointer to a `VkSurfaceKHR` handle in which the created surface is returned.

### Valid Usage (Implicit)
- **VUID-vkCreateDisplayPlaneSurfaceKHR-instance-parameter**
  - `instance` **must** be a valid `VkInstance` handle
- **VUID-vkCreateDisplayPlaneSurfaceKHR-pCreateInfo-parameter**
  - `pCreateInfo` **must** be a valid pointer to a valid `VkDisplaySurfaceCreateInfoKHR` structure
- **VUID-vkCreateDisplayPlaneSurfaceKHR-pAllocator-parameter**
  - If `pAllocator` is not NULL, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateDisplayPlaneSurfaceKHR-pSurface-parameter**
  - `pSurface` **must** be a valid pointer to a `VkSurfaceKHR` handle

### Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkDisplaySurfaceCreateInfoKHR` structure is defined as:
// Provided by VK_KHR_display

typedef struct VkDisplaySurfaceCreateInfoKHR {
    VkStructureType                   sType;
    const void*                       pNext;
    VkDisplaySurfaceCreateFlagsKHR    flags;
    VkDisplayModeKHR                  displayMode;
    uint32_t                          planeIndex;
    uint32_t                          planeStackIndex;
    VkSurfaceTransformFlagBitsKHR     transform;
    float                             globalAlpha;
    VkDisplayPlaneAlphaFlagBitsKHR    alphaMode;
    VkExtent2D                        imageExtent;
} VkDisplaySurfaceCreateInfoKHR;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **flags** is reserved for future use, and **must** be zero.
• **displayMode** is a VkDisplayModeKHR handle specifying the mode to use when displaying this surface.
• **planeIndex** is the plane on which this surface appears.
• **planeStackIndex** is the z-order of the plane.
• **transform** is a VkSurfaceTransformFlagBitsKHR value specifying the transformation to apply to images as part of the scanout operation.
• **globalAlpha** is the global alpha value. This value is ignored if **alphaMode** is not VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR.
• **alphaMode** is a VkDisplayPlaneAlphaFlagBitsKHR value specifying the type of alpha blending to use.
• **imageExtent** is the size of the presentable images to use with the surface.

**Note**

Creating a display surface **must** not modify the state of the displays, planes, or other resources it names. For example, it **must** not apply the specified mode to be set on the associated display. Application of display configuration occurs as a side effect of presenting to a display surface.
Valid Usage

- **VUID-VkDisplaySurfaceCreateInfoKHR-planeIndex-01252**
  
  `planeIndex` must be less than the number of display planes supported by the device as determined by calling `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`.

- **VUID-VkDisplaySurfaceCreateInfoKHR-planeReorderPossible-01253**
  
  If the `planeReorderPossible` member of the `VkDisplayPropertiesKHR` structure returned by `vkGetPhysicalDeviceDisplayPropertiesKHR` for the display corresponding to `displayMode` is `VK_TRUE` then `planeStackIndex` must be less than the number of display planes supported by the device as determined by calling `vkGetPhysicalDeviceDisplayPlanePropertiesKHR`; otherwise `planeStackIndex` must equal the `currentStackIndex` member of `VkDisplayPlanePropertiesKHR` returned by `vkGetPhysicalDeviceDisplayPlanePropertiesKHR` for the display plane corresponding to `displayMode`.

- **VUID-VkDisplaySurfaceCreateInfoKHR-alphaMode-01254**
  
  If `alphaMode` is `VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR` then `globalAlpha` must be between 0 and 1, inclusive.

- **VUID-VkDisplaySurfaceCreateInfoKHR-alphaMode-01255**
  
  `alphaMode` must be one of the bits present in the `supportedAlpha` member of `VkDisplayPlaneCapabilitiesKHR` for the display plane corresponding to `displayMode`.

- **VUID-VkDisplaySurfaceCreateInfoKHR-width-01256**
  
  The `width` and `height` members of `imageExtent` must be less than or equal to `VkPhysicalDeviceLimits::maxImageDimension2D`.

Valid Usage (Implicit)

- **VUID-VkDisplaySurfaceCreateInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR`.

- **VUID-VkDisplaySurfaceCreateInfoKHR-pNext-pNext**
  
  `pNext` must be `NULL`.

- **VUID-VkDisplaySurfaceCreateInfoKHR-flags-zerobitmask**
  
  `flags` must be 0.

- **VUID-VkDisplaySurfaceCreateInfoKHR-displayMode-parameter**
  
  `displayMode` must be a valid `VkDisplayModeKHR` handle.

- **VUID-VkDisplaySurfaceCreateInfoKHR-transform-parameter**
  
  `transform` must be a valid `VkSurfaceTransformFlagBitsKHR` value.

- **VUID-VkDisplaySurfaceCreateInfoKHR-alphaMode-parameter**
  
  `alphaMode` must be a valid `VkDisplayPlaneAlphaFlagBitsKHR` value.

// Provided by VK_KHR_display

typedef VkFlags VkDisplaySurfaceCreateFlagsKHR;
VkDisplaySurfaceCreateFlagsKHR is a bitmask type for setting a mask, but is currently reserved for future use.

Possible values of VkDisplaySurfaceCreateInfoKHR::alphaMode, specifying the type of alpha blending to use on a display, are:

```c
// Provided by VK_KHR_display
typedef enum VkDisplayPlaneAlphaFlagBitsKHR {
    VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR = 0x00000002,
    VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR = 0x00000004,
    VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_PREMULTIPLIED_BIT_KHR = 0x00000008,
} VkDisplayPlaneAlphaFlagBitsKHR;
```

- **VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR** specifies that the source image will be treated as opaque.
- **VK_DISPLAY_PLANE_ALPHA_GLOBAL_BIT_KHR** specifies that a global alpha value must be specified that will be applied to all pixels in the source image.
- **VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR** specifies that the alpha value will be determined by the alpha channel of the source image's pixels. If the source format contains no alpha values, no blending will be applied. The source alpha values are not premultiplied into the source image's other color channels.
- **VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_PREMULTIPLIED_BIT_KHR** is equivalent to **VK_DISPLAY_PLANE_ALPHA_PER_PIXEL_BIT_KHR**, except the source alpha values are assumed to be premultiplied into the source image's other color channels.

```c
// Provided by VK_KHR_display
typedef VkFlags VkDisplayPlaneAlphaFlagsKHR;
```

VkDisplayPlaneAlphaFlagsKHR is a bitmask type for setting a mask of zero or more VkDisplayPlaneAlphaFlagBitsKHR.

### 33.3.4. Presenting to headless surfaces

Vulkan rendering can be presented to a headless surface, where the presentation operation is a no-op producing no externally-visible result.
Note

Because there is no real presentation target, the headless presentation engine may be extended to impose an arbitrary or customisable set of restrictions and features. This makes it a useful portable test target for applications targeting a wide range of presentation engines where the actual target presentation engines might be scarce, unavailable or otherwise undesirable or inconvenient to use for general Vulkan application development.

The usual surface query mechanisms must be used to determine the actual restrictions and features of the implementation.

To create a headless VkSurfaceKHR object, call:

```c
// Provided by VK_EXT_headless_surface
VkResult vkCreateHeadlessSurfaceEXT(
    VkInstance                                  instance,
    const VkHeadlessSurfaceCreateInfoEXT*       pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSurfaceKHR*                               pSurface);
```

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to a VkHeadlessSurfaceCreateInfoEXT structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available (see Memory Allocation).
- `pSurface` is a pointer to a VkSurfaceKHR handle in which the created surface object is returned.

Valid Usage (Implicit)

- VUID-vkCreateHeadlessSurfaceEXT-instance-parameter
  `instance` must be a valid VkInstance handle

- VUID-vkCreateHeadlessSurfaceEXT-pCreateInfo-parameter
  `pCreateInfo` must be a valid pointer to a valid VkHeadlessSurfaceCreateInfoEXT structure

- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateHeadlessSurfaceEXT-pSurface-parameter
  `pSurface` must be a valid pointer to a VkSurfaceKHR handle
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkHeadlessSurfaceCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_headless_surface
typedef struct VkHeadlessSurfaceCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkHeadlessSurfaceCreateFlagsEXT flags;
} VkHeadlessSurfaceCreateInfoEXT;
```

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **flags** is reserved for future use.

Valid Usage (Implicit)

• VUID-VkHeadlessSurfaceCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT

• VUID-VkHeadlessSurfaceCreateInfoEXT-pNext-pNext
  pNext must be NULL

• VUID-VkHeadlessSurfaceCreateInfoEXT-flags-zerobitmask
  flags must be 0

For headless surfaces, **currentExtent** is the reserved value (0xFFFFFFFF, 0xFFFFFFFF). Whatever the application sets a swapchain’s **imageExtent** to will be the size of the surface, after the first image is presented.

```c
// Provided by VK_EXT_headless_surface
typedef VkFlags VkHeadlessSurfaceCreateFlagsEXT;
```

**VkHeadlessSurfaceCreateFlagsEXT** is a bitmask type for setting a mask, but is currently reserved for future use.
33.4. Querying for WSI Support

Not all physical devices will include WSI support. Within a physical device, not all queue families will support presentation. WSI support and compatibility can be determined in a platform-neutral manner (which determines support for presentation to a particular surface object) and additionally may be determined in platform-specific manners (which determine support for presentation on the specified physical device but do not guarantee support for presentation to a particular surface object).

To determine whether a queue family of a physical device supports presentation to a given surface, call:

```c
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(
    VkPhysicalDevice                            physicalDevice,
    uint32_t                                    queueFamilyIndex,
    VkSurfaceKHR                                surface,
    VkBool32*                                   pSupported);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family.
- `surface` is the surface.
- `pSupported` is a pointer to a `VkBool32`, which is set to `VK_TRUE` to indicate support, and `VK_FALSE` otherwise.

**Valid Usage**

- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-queueFamilyIndex-01269
  
  `queueFamilyIndex` must be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-physicalDevice-parameter
  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-surface-parameter
  
  `surface` must be a valid `VkSurfaceKHR` handle

- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-pSupported-parameter
  
  `pSupported` must be a valid pointer to a `VkBool32` value

- VUID-vkGetPhysicalDeviceSurfaceSupportKHR-commonparent
  
  Both of `physicalDevice`, and `surface` must have been created, allocated, or retrieved from the same `VkInstance`
### Return Codes

**Success**
- VK_SUCCESS

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

### 33.4.1. Android Platform

On Android, all physical devices and queue families must be capable of presentation with any native window. As a result there is no Android-specific query for these capabilities.

### 33.4.2. Wayland Platform

To determine whether a queue family of a physical device supports presentation to a Wayland compositor, call:

```c
// Provided by VK_KHR_wayland_surface
VkBool32 vkGetPhysicalDeviceWaylandPresentationSupportKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    struct wl_display* display);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `display` is a pointer to the `wl_display` associated with a Wayland compositor.

This platform-specific function can be called prior to creating a surface.

### Valid Usage

- VUID-vkGetPhysicalDeviceWaylandPresentationSupportKHR-queueFamilyIndex-01306
  queueFamilyIndex must be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`
33.4.3. Win32 Platform

To determine whether a queue family of a physical device supports presentation to the Microsoft Windows desktop, call:

```c
// Provided by VK_KHR_win32_surface
VkBool32 vkGetPhysicalDeviceWin32PresentationSupportKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.

This platform-specific function can be called prior to creating a surface.

33.4.4. XCB Platform

To determine whether a queue family of a physical device supports presentation to an X11 server, using the XCB client-side library, call:
Provided by VK_KHR_xcb_surface

```c
VkBool32 vkGetPhysicalDeviceXcbPresentationSupportKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    xcb_connection_t* connection,
    xcb_visualid_t visual_id);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `connection` is a pointer to an `xcb_connection_t` to the X server.
- `visual_id` is an X11 visual (`xcb_visualid_t`).

This platform-specific function can be called prior to creating a surface.

**Valid Usage**

- `VUID-vkGetPhysicalDeviceXcbPresentationSupportKHR-queueFamilyIndex-01312`
  - `queueFamilyIndex` must be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`

**Valid Usage (Implicit)**

- `VUID-vkGetPhysicalDeviceXcbPresentationSupportKHR-physicalDevice-parameter`
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `VUID-vkGetPhysicalDeviceXcbPresentationSupportKHR-connection-parameter`
  - `connection` must be a valid pointer to an `xcb_connection_t` value

### 33.4.5. Xlib Platform

To determine whether a queue family of a physical device supports presentation to an X11 server, using the Xlib client-side library, call:

```c
// Provided by VK_KHR_xlib_surface
VkBool32 vkGetPhysicalDeviceXlibPresentationSupportKHR(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    Display* dpy,
    VisualID visualID);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `dpy` is a pointer to an Xlib `Display` connection to the server.
• `visualId` is an X11 visual (`VisualID`).

This platform-specific function can be called prior to creating a surface.

### Valid Usage

- VUID-vkGetPhysicalDeviceXlibPresentationSupportKHR-queueFamilyIndex-01315
  queueFamilyIndex must be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceXlibPresentationSupportKHR-physicalDevice-parameter
  physicalDevice must be a valid `VkPhysicalDevice` handle.
- VUID-vkGetPhysicalDeviceXlibPresentationSupportKHR-dpy-parameter
  `dpy` must be a valid pointer to a `Display` value.

### 33.4.6. DirectFB Platform

To determine whether a queue family of a physical device supports presentation with DirectFB library, call:

```c
// Provided by VK_EXT_directfb_surface
VkBool32 vkGetPhysicalDeviceDirectFBPresentationSupportEXT(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    IDirectFB* dfb);
```

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family index.
- `dfb` is a pointer to the `IDirectFB` main interface of DirectFB.

This platform-specific function can be called prior to creating a surface.

### Valid Usage

- VUID-vkGetPhysicalDeviceDirectFBPresentationSupportEXT-queueFamilyIndex-04119
  queueFamilyIndex must be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the given `physicalDevice`.
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceDirectFBPresentationSupportEXT-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceDirectFBPresentationSupportEXT-dfb-parameter
dfb must be a valid pointer to an IDirectFB value

33.4.7. Fuchsia Platform

On Fuchsia, all physical devices and queue families must be capable of presentation with any ImagePipe. As a result there is no Fuchsia-specific query for these capabilities.

33.4.8. Google Games Platform

On Google Games Platform, all physical devices and queue families with the VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT capabilities must be capable of presentation with any Google Games Platform stream descriptor. As a result, there is no query specific to Google Games Platform for these capabilities.

33.4.9. iOS Platform

On iOS, all physical devices and queue families must be capable of presentation with any layer. As a result there is no iOS-specific query for these capabilities.

33.4.10. macOS Platform

On macOS, all physical devices and queue families must be capable of presentation with any layer. As a result there is no macOS-specific query for these capabilities.

33.4.11. VI Platform

On VI, all physical devices and queue families must be capable of presentation with any layer. As a result there is no VI-specific query for these capabilities.

33.4.12. QNX Screen Platform

To determine whether a queue family of a physical device supports presentation to a QNX Screen compositor, call:

```c
// Provided by VK_QNX_screen_surface
VkBool32 vkGetPhysicalDeviceScreenPresentationSupportQNX(
    VkPhysicalDevice physicalDevice,
    uint32_t queueFamilyIndex,
    struct _screen_window* window);
```

- physicalDevice is the physical device.
• queueFamilyIndex is the queue family index.
• window is the QNX Screen window object.

This platform-specific function can be called prior to creating a surface.

### Valid Usage

- VUID-vkGetPhysicalDeviceScreenPresentationSupportQNX-queueFamilyIndex-04743
  queueFamilyIndex must be less than pQueueFamilyPropertyCount returned by vkGetPhysicalDeviceQueueFamilyProperties for the given physicalDevice

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceScreenPresentationSupportQNX-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle
- VUID-vkGetPhysicalDeviceScreenPresentationSupportQNX-window-parameter
  window must be a valid pointer to a _screen_window value

### 33.5. Surface Queries

The capabilities of a swapchain targeting a surface are the intersection of the capabilities of the WSI platform, the native window or display, and the physical device. The resulting capabilities can be obtained with the queries listed below in this section.

**Note**
In addition to the surface capabilities as obtained by surface queries below, swapchain images are also subject to ordinary image creation limits as reported by vkGetPhysicalDeviceImageFormatProperties. As an application is instructed by the appropriate Valid Usage sections, both the surface capabilities and the image creation limits have to be satisfied whenever swapchain images are created.

### 33.5.1. Surface Capabilities

To query the basic capabilities of a surface, needed in order to create a swapchain, call:

```c
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfaceCapabilitiesKHR(  
    VkPhysicalDevice                            physicalDevice,  
    VkSurfaceKHR                                surface,  
    VkSurfaceCapabilitiesKHR*                   pSurfaceCapabilities);
```

- physicalDevice is the physical device that will be associated with the swapchain to be created, as described for vkCreateSwapchainKHR.
- `surface` is the surface that will be associated with the swapchain.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilitiesKHR` structure in which the capabilities are returned.

### Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-physicalDevice-parameter`  
  `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- `VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-surface-parameter`  
  `surface` **must** be a valid `VkSurfaceKHR` handle
- `VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-pSurfaceCapabilities-parameter`  
  `pSurfaceCapabilities` **must** be a valid pointer to a `VkSurfaceCapabilitiesKHR` structure
- `VUID-vkGetPhysicalDeviceSurfaceCapabilitiesKHR-commonparent`  
  Both of `physicalDevice`, and `surface` **must** have been created, allocated, or retrieved from the same `VkInstance`

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkSurfaceCapabilitiesKHR` structure is defined as:

```c
// Provided by VK_KHR_surface
typedef struct VkSurfaceCapabilitiesKHR {
    uint32_t minImageCount;
    uint32_t maxImageCount;
    VkExtent2D currentExtent;
    VkExtent2D minImageExtent;
    VkExtent2D maxImageExtent;
    uint32_t maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkSurfaceTransformFlagBitsKHR currentTransform;
    VkCompositeAlphaFlagsKHR supportedCompositeAlpha;
    VkImageUsageFlags supportedUsageFlags;
} VkSurfaceCapabilitiesKHR;
```

- `minImageCount` is the minimum number of images the specified device supports for a swapchain
created for the surface, and will be at least one.

- `maxImageCount` is the maximum number of images the specified device supports for a swapchain created for the surface, and will be either 0, or greater than or equal to `minImageCount`. A value of 0 means that there is no limit on the number of images, though there may be limits related to the total amount of memory used by presentable images.

- `currentExtent` is the current width and height of the surface, or the special value (0xFFFFFFFF, 0xFFFFFFFF) indicating that the surface size will be determined by the extent of a swapchain targeting the surface.

- `minImageExtent` contains the smallest valid swapchain extent for the surface on the specified device. The `width` and `height` of the extent will each be less than or equal to the corresponding `width` and `height` of `currentExtent`, unless `currentExtent` has the special value described above.

- `maxImageExtent` contains the largest valid swapchain extent for the surface on the specified device. The `width` and `height` of the extent will each be greater than or equal to the corresponding `width` and `height` of `minImageExtent`. The `width` and `height` of the extent will each be greater than or equal to the corresponding `width` and `height` of `currentExtent`, unless `currentExtent` has the special value described above.

- `maxImageArrayLayers` is the maximum number of layers presentable images can have for a swapchain created for this device and surface, and will be at least one.

- `supportedTransforms` is a bitmask of `VkSurfaceTransformFlagBitsKHR` indicating the presentation transforms supported for the surface on the specified device. At least one bit will be set.

- `currentTransform` is `VkSurfaceTransformFlagBitsKHR` value indicating the surface's current transform relative to the presentation engine's natural orientation.

- `supportedCompositeAlpha` is a bitmask of `VkCompositeAlphaFlagBitsKHR`, representing the alpha compositing modes supported by the presentation engine for the surface on the specified device, and at least one bit will be set. Opaque composition can be achieved in any alpha compositing mode by either using an image format that has no alpha component, or by ensuring that all pixels in the presentable images have an alpha value of 1.0.

- `supportedUsageFlags` is a bitmask of `VkImageUsageFlagBits` representing the ways the application can use the presentable images of a swapchain created with `VkPresentModeKHR` set to `VK_PRESENT_MODE_IMMEDIATE_KHR`, `VK_PRESENT_MODE_MAILBOX_KHR`, `VK_PRESENT_MODE_FIFO_KHR` or `VK_PRESENT_MODE_FIFO_RELAXED_KHR` for the surface on the specified device. `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` must be included in the set but implementations may support additional usages.

---

**Note**

Supported usage flags of a presentable image when using `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` presentation mode are provided by `VkSharedPresentSurfaceCapabilitiesKHR::sharedPresentSupportedUsageFlags`. 

---
Note
Formulas such as min(N, maxImageCount) are not correct, since maxImageCount may be zero.

To query the basic capabilities of a surface defined by the core or extensions, call:

```
// Provided by VK_KHR_get_surface_capabilities2
VkResult vkGetPhysicalDeviceSurfaceCapabilities2KHR(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,
    VkSurfaceCapabilities2KHR* pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `pSurfaceInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by `vkCreateSwapchainKHR`.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilities2KHR` structure in which the capabilities are returned.

`vkGetPhysicalDeviceSurfaceCapabilities2KHR` behaves similarly to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with the ability to specify extended inputs via chained input structures, and to return extended information via chained output structures.

Valid Usage

- `VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-pNext-02671` If a `VkSurfaceCapabilitiesFullScreenExclusiveEXT` structure is included in the `pNext` chain of `pSurfaceCapabilities`, a `VkSurfaceFullScreenExclusiveWin32InfoEXT` structure must be included in the `pNext` chain of `pSurfaceInfo`.

Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-physicalDevice-parameter` `physicalDevice` must be a valid `VkPhysicalDevice` handle.
- `VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-pSurfaceInfo-parameter` `pSurfaceInfo` must be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure.
- `VUID-vkGetPhysicalDeviceSurfaceCapabilities2KHR-pSurfaceCapabilities-parameter` `pSurfaceCapabilities` must be a valid pointer to a `VkSurfaceCapabilities2KHR` structure.
Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

The `VkPhysicalDeviceSurfaceInfo2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_surface_capabilities2
typedef struct VkPhysicalDeviceSurfaceInfo2KHR {
    VkStructureType sType;
    const void* pNext;
    VkSurfaceKHR surface;
} VkPhysicalDeviceSurfaceInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `surface` is the surface that will be associated with the swapchain.

The members of `VkPhysicalDeviceSurfaceInfo2KHR` correspond to the arguments to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with `sType` and `pNext` added for extensibility.

Additional capabilities of a surface may be available to swapchains created with different full-screen exclusive settings - particularly if exclusive full-screen access is application controlled. These additional capabilities can be queried by adding a `VkSurfaceFullScreenExclusiveInfoEXT` structure to the `pNext` chain of this structure when used to query surface properties. Additionally, for Win32 surfaces with application controlled exclusive full-screen access, chaining a `VkSurfaceFullScreenExclusiveWin32InfoEXT` structure may also report additional surface capabilities. These additional capabilities only apply to swapchains created with the same parameters included in the `pNext` chain of `VkSwapchainCreateInfoKHR`.

Valid Usage

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-pNext-02672

If the `pNext` chain includes a `VkSurfaceFullScreenExclusiveInfoEXT` structure with its `fullScreenExclusive` member set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`, and `surface` was created using `vkCreateWin32SurfaceKHR`, a `VkSurfaceFullScreenExclusiveWin32InfoEXT` structure must be included in the `pNext` chain.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR`

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-pNext-pNext
  
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkSurfaceFullScreenExclusiveInfoEXT` or `VkSurfaceFullScreenExclusiveWin32InfoEXT`

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-sType-unique
  
  The `sType` value of each struct in the `pNext` chain must be unique

- VUID-VkPhysicalDeviceSurfaceInfo2KHR-surface-parameter
  
  `surface` must be a valid `VkSurfaceKHR` handle

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkSurfaceFullScreenExclusiveInfoEXT` structure, then that structure specifies the application's preferred full-screen transition behavior.

The `VkSurfaceFullScreenExclusiveInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_full_screen_exclusive
typedef struct VkSurfaceFullScreenExclusiveInfoEXT {
    VkStructureType             sType;
    void*                       pNext;
    VkFullScreenExclusiveEXT    fullScreenExclusive;
} VkSurfaceFullScreenExclusiveInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `fullScreenExclusive` is a `VkFullScreenExclusiveEXT` value specifying the preferred full-screen transition behavior.

If this structure is not present, `fullScreenExclusive` is considered to be `VK_FULLSCREEN_EXCLUSIVE_DEFAULT_EXT`.

Valid Usage (Implicit)

- VUID-VkSurfaceFullScreenExclusiveInfoEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_SURFACE_FULLSCREEN_EXCLUSIVE_INFO_EXT`

- VUID-VkSurfaceFullScreenExclusiveInfoEXT-fullScreenExclusive-parameter
  
  `fullScreenExclusive` must be a valid `VkFullScreenExclusiveEXT` value

Possible values of `VkSurfaceFullScreenExclusiveInfoEXT::fullScreenExclusive` are:
typedef enum VkFullScreenExclusiveEXT {
    VK_FULL_SCREEN_EXCLUSIVE_DEFAULT_EXT = 0,
    VK_FULL_SCREEN_EXCLUSIVE_ALLOWED_EXT = 1,
    VK_FULL_SCREEN_EXCLUSIVE_DISALLOWED_EXT = 2,
    VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT = 3,
} VkFullScreenExclusiveEXT;

- VK_FULL_SCREEN_EXCLUSIVE_DEFAULT_EXT indicates the implementation should determine the appropriate full-screen method by whatever means it deems appropriate.

- VK_FULL_SCREEN_EXCLUSIVE_ALLOWED_EXT indicates the implementation may use full-screen exclusive mechanisms when available. Such mechanisms may result in better performance and/or the availability of different presentation capabilities, but may require a more disruptive transition during swapchain initialization, first presentation and/or destruction.

- VK_FULL_SCREEN_EXCLUSIVE_DISALLOWED_EXT indicates the implementation should avoid using full-screen mechanisms which rely on disruptive transitions.

- VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT indicates the application will manage full-screen exclusive mode by using the vkAcquireFullScreenExclusiveModeEXT and vkReleaseFullScreenExclusiveModeEXT commands.

The VkSurfaceFullScreenExclusiveWin32InfoEXT structure is defined as:

typedef struct VkSurfaceFullScreenExclusiveWin32InfoEXT {
    VkStructureType sType;
    const void* pNext;
    HMONITOR hmonitor;
} VkSurfaceFullScreenExclusiveWin32InfoEXT;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- hmonitor is the Win32 HMONITOR handle identifying the display to create the surface with.

Note

If hmonitor is invalidated (e.g. the monitor is unplugged) during the lifetime of a swapchain created with this structure, operations on that swapchain will return VK_ERROR_OUT_OF_DATE_KHR.

Note

It is the responsibility of the application to change the display settings of the targeted Win32 display using the appropriate platform APIs. Such changes may alter the surface capabilities reported for the created surface.
Valid Usage

- VUID-VkSurfaceFullScreenExclusiveWin32InfoEXT-hmonitor-02673
  \textit{hmonitor must be a valid HMONITOR}

Valid Usage (Implicit)

- VUID-VkSurfaceFullScreenExclusiveWin32InfoEXT-sType-sType
  \textit{sType must be VK_STRUCTURE_TYPE_SURFACE_FULLSCREEN_EXCLUSIVE_WIN32_INFO_EXT}

The \texttt{VkSurfaceCapabilities2KHR} structure is defined as:

```c
typedef struct VkSurfaceCapabilities2KHR {
    VkStructureType             sType;
    void*                       pNext;
    VkSurfaceCapabilitiesKHR    surfaceCapabilities;
} VkSurfaceCapabilities2KHR;
```

- \textit{sType} is the type of this structure.
- \textit{pNext} is NULL or a pointer to a structure extending this structure.
- \textit{surfaceCapabilities} is a \texttt{VkSurfaceCapabilitiesKHR} structure describing the capabilities of the specified surface.

Valid Usage (Implicit)

- VUID-VkSurfaceCapabilities2KHR-sType-sType
  \textit{sType must be VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR}

- VUID-VkSurfaceCapabilities2KHR-pNext-pNext
  Each \textit{pNext} member of any structure (including this one) in the \textit{pNext} chain must be either NULL or a pointer to a valid instance of \texttt{VkDisplayNativeHdrSurfaceCapabilitiesAMD}, \texttt{VkSharedPresentSurfaceCapabilitiesKHR}, \texttt{VkSurfaceCapabilitiesFullScreenExclusiveEXT}, or \texttt{VkSurfaceProtectedCapabilitiesKHR}

- VUID-VkSurfaceCapabilities2KHR-sType-unique
  The \textit{sType} value of each struct in the \textit{pNext} chain must be unique

An application queries if a protected \texttt{VkSurfaceKHR} is displayable on a specific windowing system using \texttt{VkSurfaceProtectedCapabilitiesKHR}, which can be passed in \textit{pNext} parameter of \texttt{VkSurfaceCapabilities2KHR}.

The \texttt{VkSurfaceProtectedCapabilitiesKHR} structure is defined as:
```c
typedef struct VkSurfaceProtectedCapabilitiesKHR {
    VkStructureType sType;
    const void* pNext;
    VkBool32 supportsProtected;
} VkSurfaceProtectedCapabilitiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `supportsProtected` specifies whether a protected swapchain created from `VkPhysicalDeviceSurfaceInfo2KHR::surface` for a particular windowing system can be displayed on screen or not. If `supportsProtected` is `VK_TRUE`, then creation of swapchains with the `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR` flag set must be supported for `surface`.

**Valid Usage (Implicit)**

- VUID-VkSurfaceProtectedCapabilitiesKHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_SURFACE_PROTECTED_CAPABILITIES_KHR`

The `VkSharedPresentSurfaceCapabilitiesKHR` structure is defined as:

```c
typedef struct VkSharedPresentSurfaceCapabilitiesKHR {
    VkStructureType sType;
    void* pNext;
    VkImageUsageFlags sharedPresentSupportedUsageFlags;
} VkSharedPresentSurfaceCapabilitiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `sharedPresentSupportedUsageFlags` is a bitmask of `VkImageUsageFlagBits` representing the ways the application can use the shared presentable image from a swapchain created with `VkPresentModeKHR` set to `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR` for the surface on the specified device. `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` must be included in the set but implementations may support additional usages.

**Valid Usage (Implicit)**

- VUID-VkSharedPresentSurfaceCapabilitiesKHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR`

The `VkDisplayNativeHdrSurfaceCapabilitiesAMD` structure is defined as:
```c
typedef struct VkDisplayNativeHdrSurfaceCapabilitiesAMD {
    VkStructureType sType;
    void* pNext;
    VkBool32 localDimmingSupport;
} VkDisplayNativeHdrSurfaceCapabilitiesAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `localDimmingSupport` specifies whether the surface supports local dimming. If this is `VK_TRUE`, `VkSwapchainDisplayNativeHdrCreateInfoAMD` can be used to explicitly enable or disable local dimming for the surface. Local dimming may also be overridden by `vkSetLocalDimmingAMD` during the lifetime of the swapchain.

### Valid Usage (Implicit)

- `VUID-VkDisplayNativeHdrSurfaceCapabilitiesAMD-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_NATIVE_HDR_SURFACE_CAPABILITIES_AMD`

The `VkSurfaceCapabilitiesFullScreenExclusiveEXT` structure is defined as:

```c
typedef struct VkSurfaceCapabilitiesFullScreenExclusiveEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 fullScreenExclusiveSupported;
} VkSurfaceCapabilitiesFullScreenExclusiveEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `fullScreenExclusiveControlSupported` is a boolean describing whether the surface is able to make use of exclusive full-screen access.

This structure can be included in the `pNext` chain of `VkSurfaceCapabilities2KHR` to determine support for exclusive full-screen access. If `fullScreenExclusiveSupported` is `VK_FALSE`, it indicates that exclusive full-screen access is not obtainable for this surface.

Applications must not attempt to create swapchains with `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT` set if `fullScreenExclusiveSupported` is `VK_FALSE`. 
Valid Usage (Implicit)

- VUID-VkSurfaceCapabilitiesFullScreenExclusiveEXT-sType-sType

  sType must be VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_FULL_SCREEN_EXCLUSIVE_EXT

To query the basic capabilities of a surface, needed in order to create a swapchain, call:

```c
// Provided by VK_EXT_display_surface_counter
vkGetPhysicalDeviceSurfaceCapabilities2EXT(
    VkPhysicalDevice                            physicalDevice,
    VkSurfaceKHR                                surface,
    VkSurfaceCapabilities2EXT*                  pSurfaceCapabilities);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `surface` is the surface that will be associated with the swapchain.
- `pSurfaceCapabilities` is a pointer to a `VkSurfaceCapabilities2EXT` structure in which the capabilities are returned.

`vkGetPhysicalDeviceSurfaceCapabilities2EXT` behaves similarly to `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, with the ability to return extended information by adding extending structures to the `pNext` chain of its `pSurfaceCapabilities` parameter.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-physicalDevice-parameter

  physicalDevice must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-surface-parameter

  surface must be a valid `VkSurfaceKHR` handle

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-pSurfaceCapabilities-parameter

  pSurfaceCapabilities must be a valid pointer to a `VkSurfaceCapabilities2EXT` structure

- VUID-vkGetPhysicalDeviceSurfaceCapabilities2EXT-commonparent

  Both of physicalDevice, and surface must have been created, allocated, or retrieved from the same `VkInstance`
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_SURFACE_LOST_KHR

The `VkSurfaceCapabilities2EXT` structure is defined as:

```c
// Provided by VK_EXT_display_surface_counter
typedef struct VkSurfaceCapabilities2EXT {
    VkStructureType sType;
    void* pNext;
    uint32_t minImageCount;
    uint32_t maxImageCount;
    VkExtent2D currentExtent;
    VkExtent2D minImageExtent;
    VkExtent2D maxImageExtent;
    uint32_t maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR supportedTransforms;
    VkSurfaceTransformFlagBitsKHR currentTransform;
    VkCompositeAlphaFlagsKHR supportedCompositeAlpha;
    VkImageUsageFlags supportedUsageFlags;
    VkSurfaceCounterFlagsEXT supportedSurfaceCounters;
} VkSurfaceCapabilities2EXT;
```

All members of `VkSurfaceCapabilities2EXT` are identical to the corresponding members of `VkSurfaceCapabilitiesKHR` where one exists. The remaining members are:

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `supportedSurfaceCounters` is a bitmask of `VkSurfaceCounterFlagBitsEXT` indicating the supported surface counter types.

Valid Usage

• VUID-VkSurfaceCapabilities2EXT-supportedSurfaceCounters-01246
  `supportedSurfaceCounters` must not include `VK_SURFACE_COUNTER_VBLANK_BIT_EXT` unless the surface queried is a display surface
Valid Usage (Implicit)

- **VUID-VkSurfaceCapabilities2EXT-sType-sType**
  
  `sType` must be **VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT**

- **VUID-VkSurfaceCapabilities2EXT-pNext-pNext**
  
  `pNext` must be **NULL**

Bits which **can** be set in `VkSurfaceCapabilities2EXT::supportedSurfaceCounters`, indicating supported surface counter types, are:

```c
// Provided by VK_EXT_display_surface_counter
typedef enum VkSurfaceCounterFlagBitsEXT {
    VK_SURFACE_COUNTER_VBLANK_BIT_EXT = 0x00000001,
    VK_SURFACE_COUNTER_VBLANK_EXT = VK_SURFACE_COUNTER_VBLANK_BIT_EXT,
} VkSurfaceCounterFlagBitsEXT;
```

- **VK_SURFACE_COUNTER_VBLANK_BIT_EXT** specifies a counter incrementing once every time a vertical blanking period occurs on the display associated with the surface.

```c
// Provided by VK_EXT_display_surface_counter
typedef VkFlags VkSurfaceCounterFlagsEXT;
```

`VkSurfaceCounterFlagsEXT` is a bitmask type for setting a mask of zero or more `VkSurfaceCounterFlagBitsEXT`.

Bits which **may** be set in `VkSurfaceCapabilitiesKHR::supportedTransforms` indicating the presentation transforms supported for the surface on the specified device, and possible values of `VkSurfaceCapabilitiesKHR::currentTransform` indicating the surface's current transform relative to the presentation engine's natural orientation, are:

```c
// Provided by VK_KHR_surface
typedef enum VkSurfaceTransformFlagBitsKHR {
    VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR = 0x00000001,
    VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR = 0x00000002,
    VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR = 0x00000004,
    VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR = 0x00000008,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR = 0x00000010,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR = 0x00000020,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR = 0x00000040,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR = 0x00000080,
    VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR = 0x00000100,
} VkSurfaceTransformFlagBitsKHR;
```

- **VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR** specifies that image content is presented without being transformed.
• **VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR** specifies that image content is rotated 90 degrees clockwise.

• **VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR** specifies that image content is rotated 180 degrees clockwise.

• **VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR** specifies that image content is rotated 270 degrees clockwise.

• **VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR** specifies that image content is mirrored horizontally.

• **VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR** specifies that image content is mirrored horizontally, then rotated 90 degrees clockwise.

• **VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR** specifies that image content is mirrored horizontally, then rotated 180 degrees clockwise.

• **VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR** specifies that image content is mirrored horizontally, then rotated 270 degrees clockwise.

• **VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR** specifies that the presentation transform is not specified, and is instead determined by platform-specific considerations and mechanisms outside Vulkan.

```c
// Provided by VK_KHR_display
typedef VkFlags VkSurfaceTransformFlagsKHR;
```

`VkSurfaceTransformFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSurfaceTransformFlagBitsKHR`.

The **supportedCompositeAlpha** member is of type `VkCompositeAlphaFlagBitsKHR`, which contains the following values:

```c
// Provided by VK_KHR_surface
typedef enum VkCompositeAlphaFlagBitsKHR {
    VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR = 0x00000002,
    VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR = 0x00000004,
    VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR = 0x00000008,
} VkCompositeAlphaFlagBitsKHR;
```

These values are described as follows:

• **VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR**: The alpha channel, if it exists, of the images is ignored in the compositing process. Instead, the image is treated as if it has a constant alpha of 1.0.

• **VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR**: The alpha channel, if it exists, of the images is respected in the compositing process. The non-alpha channels of the image are expected to already be multiplied by the alpha channel by the application.

• **VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR**: The alpha channel, if it exists, of the images is respected in the compositing process. The non-alpha channels of the image are not expected to
already be multiplied by the alpha channel by the application; instead, the compositor will multiply the non-alpha channels of the image by the alpha channel during compositing.

- **VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR**: The way in which the presentation engine treats the alpha channel in the images is unknown to the Vulkan API. Instead, the application is responsible for setting the composite alpha blending mode using native window system commands. If the application does not set the blending mode using native window system commands, then a platform-specific default will be used.

```c
// Provided by VK_KHR_surface
typedef VkFlags VkCompositeAlphaFlagsKHR;
```

`VkCompositeAlphaFlagsKHR` is a bitmask type for setting a mask of zero or more `VkCompositeAlphaFlagBitsKHR`.

### 33.5.2. Surface Format Support

To query the supported swapchain format-color space pairs for a surface, call:

```c
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfaceFormatsKHR(
    VkPhysicalDevice physicalDevice,  // Physical device associated with the swapchain to be created.
    VkSurfaceKHR surface,            // Surface associated with the swapchain.
    uint32_t* pSurfaceFormatCount,   // Pointer to an integer related to the number of format pairs available or queried.
    VkSurfaceFormatKHR* pSurfaceFormats);
```

- **physicalDevice** is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- **surface** is the surface that will be associated with the swapchain.
- **pSurfaceFormatCount** is a pointer to an integer related to the number of format pairs available or queried, as described below.
- **pSurfaceFormats** is either `NULL` or a pointer to an array of `VkSurfaceFormatKHR` structures.

If `pSurfaceFormats` is `NULL`, then the number of format pairs supported for the given `surface` is returned in `pSurfaceFormatCount`. Otherwise, `pSurfaceFormatCount` must point to a variable set by the user to the number of elements in the `pSurfaceFormats` array, and on return the variable is overwritten with the number of structures actually written to `pSurfaceFormats`. If the value of `pSurfaceFormatCount` is less than the number of format pairs supported, at most `pSurfaceFormatCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available format pairs were returned.

The number of format pairs supported must be greater than or equal to 1. `pSurfaceFormats` must not contain an entry whose value for **format** is `VK_FORMAT_UNDEFINED`.

If `pSurfaceFormats` includes an entry whose value for **colorSpace** is `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR` and whose value for **format** is a UNORM (or SRGB) format and the corresponding SRGB (or UNORM) format is a color renderable format for
VK_IMAGE_TILING_OPTIMAL, then pSurfaceFormats must also contain an entry with the same value for colorSpace and format equal to the corresponding SRGB (or UNORM) format.

Valid Usage

• VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-surface-02739
  surface must be supported by physicalDevice, as reported by vkGetPhysicalDeviceSurfaceSupportKHR or an equivalent platform-specific mechanism

Valid Usage (Implicit)

• VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

• VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-surface-parameter
  surface must be a valid VkSurfaceKHR handle

• VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-pSurfaceFormatCount-parameter
  pSurfaceFormatCount must be a valid pointer to a uint32_t value

• VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-pSurfaceFormats-parameter
  If the value referenced by pSurfaceFormatCount is not 0, and pSurfaceFormats is not NULL, pSurfaceFormats must be a valid pointer to an array of pSurfaceFormatCount VkSurfaceFormatKHR structures

• VUID-vkGetPhysicalDeviceSurfaceFormatsKHR-commonparent
  Both of physicalDevice, and surface must have been created, allocated, or retrieved from the same VkInstance

Return Codes

Success

• VK_SUCCESS
• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_SURFACE_LOST_KHR

The VkSurfaceFormatKHR structure is defined as:
```c
// Provided by VK_KHR_surface
typedef struct VkSurfaceFormatKHR {
    VkFormat format;
    VkColorSpaceKHR colorSpace;
} VkSurfaceFormatKHR;
```

- `format` is a `VkFormat` that is compatible with the specified surface.
- `colorSpace` is a presentation `VkColorSpaceKHR` that is compatible with the surface.

To query the supported swapchain format tuples for a surface, call:

```c
// Provided by VK_KHR_get_surface_capabilities2
VkResult vkGetPhysicalDeviceSurfaceFormats2KHR(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,
    uint32_t* pSurfaceFormatCount,
    VkSurfaceFormat2KHR* pSurfaceFormats);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `pSurfaceInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by `vkCreateSwapchainKHR`.
- `pSurfaceFormatCount` is a pointer to an integer related to the number of format tuples available or queried, as described below.
- `pSurfaceFormats` is either `NULL` or a pointer to an array of `VkSurfaceFormat2KHR` structures.

`vkGetPhysicalDeviceSurfaceFormats2KHR` behaves similarly to `vkGetPhysicalDeviceSurfaceFormatsKHR`, with the ability to be extended via `pNext` chains.

If `pSurfaceFormats` is `NULL`, then the number of format tuples supported for the given `surface` is returned in `pSurfaceFormatCount`. Otherwise, `pSurfaceFormatCount` must point to a variable set by the user to the number of elements in the `pSurfaceFormats` array, and on return the variable is overwritten with the number of structures actually written to `pSurfaceFormats`. If the value of `pSurfaceFormatCount` is less than the number of format tuples supported, at most `pSurfaceFormatCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available values were returned.

**Valid Usage**

- VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceInfo-02740
  `pSurfaceInfo->surface` must be supported by `physicalDevice`, as reported by `vkGetPhysicalDeviceSurfaceSupportKHR` or an equivalent platform-specific mechanism
Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-physicalDevice-parameter**
  The `physicalDevice` must be a valid `VkPhysicalDevice` handle.

- **VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceInfo-parameter**
  The `pSurfaceInfo` must be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure.

- **VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceFormatCount-parameter**
  The `pSurfaceFormatCount` must be a valid pointer to a `uint32_t` value.

- **VUID-vkGetPhysicalDeviceSurfaceFormats2KHR-pSurfaceFormats-parameter**
  If the value referenced by `pSurfaceFormatCount` is not 0, and `pSurfaceFormats` is not NULL, `pSurfaceFormats` must be a valid pointer to an array of `VkSurfaceFormat2KHR` structures.

Return Codes

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

The `VkSurfaceFormat2KHR` structure is defined as:

```c
// Provided by VK_KHR_get_surface_capabilities2
typedef struct VkSurfaceFormat2KHR {
    VkStructureType       sType;
    void*                 pNext;
    VkSurfaceFormatKHR    surfaceFormat;
} VkSurfaceFormat2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `surfaceFormat` is a `VkSurfaceFormatKHR` structure describing a format-color space pair that is compatible with the specified surface.
Valid Usage (Implicit)

- **VUID-VkSurfaceFormat2KHR-sType-sType**
  
  **sType** **must** be **VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR**

- **VUID-VkSurfaceFormat2KHR-pNext-pNext**
  
  **pNext** **must** be **NULL**

While the *format* of a presentable image refers to the encoding of each pixel, the *colorSpace* determines how the presentation engine interprets the pixel values. A color space in this document refers to a specific color space (defined by the chromaticities of its primaries and a white point in CIE Lab), and a transfer function that is applied before storing or transmitting color data in the given color space.

Possible values of `VkSurfaceFormatKHR::colorSpace`, specifying supported color spaces of a presentation engine, are:
// Provided by VK_KHR_surface
typedef enum VkColorSpaceKHR {
    VK_COLOR_SPACE_SRGB_NONLINEAR_KHR = 0,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT = 1000104001,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT = 1000104002,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT = 1000104003,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT = 1000104004,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_BT709_LINEAR_EXT = 1000104005,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_BT709_NONLINEAR_EXT = 1000104006,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_BT2020_LINEAR_EXT = 1000104007,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_HDR10_ST2084_EXT = 1000104008,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_DOLBYVISION_EXT = 1000104009,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_HDR10_HLG_EXT = 1000104010,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT = 1000104011,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT = 1000104012,
    // Provided by VK_EXT_swapchain_colorspace
    VK_COLOR_SPACE_PASS_THROUGH_EXT = 1000104013,
    // Provided by VK_AMD_display_native_hdr
    VK_COLORSPACE_SRGB_NONLINEAR_KHR = VK_COLOR_SPACE_SRGB_NONLINEAR_KHR,
    VK_COLOR_SPACE_DCI_P3_LINEAR_EXT = VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT,
} VkColorSpaceKHR;

- **VK_COLOR_SPACE_SRGB_NONLINEAR_KHR** specifies support for the sRGB color space.
- **VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT** specifies support for the Display-P3 color space to be displayed using an sRGB-like EOTF (defined below).
- **VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT** specifies support for the extended sRGB color space to be displayed using a linear EOTF.
- **VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT** specifies support for the extended sRGB color space to be displayed using an sRGB EOTF.
- **VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT** specifies support for the Display-P3 color space to be displayed using a linear EOTF.
- **VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT** specifies support for the DCI-P3 color space to be displayed...
using the DCI-P3 EOTF. Note that values in such an image are interpreted as XYZ encoded color
data by the presentation engine.

- **VK_COLOR_SPACE_BT709_LINEAR_EXT** specifies support for the BT709 color space to be displayed
  using a linear EOTF.

- **VK_COLOR_SPACE_BT709_NONLINEAR_EXT** specifies support for the BT709 color space to be displayed
  using the SMPTE 170M EOTF.

- **VK_COLOR_SPACE_BT2020_LINEAR_EXT** specifies support for the BT2020 color space to be displayed
  using a linear EOTF.

- **VK_COLOR_SPACE_HDR10_ST2084_EXT** specifies support for the HDR10 (BT2020 color) space to be
  displayed using the SMPTE ST2084 Perceptual Quantizer (PQ) EOTF.

- **VK_COLOR_SPACE_DOLBYVISION_EXT** specifies support for the Dolby Vision (BT2020 color space),
  proprietary encoding, to be displayed using the SMPTE ST2084 EOTF.

- **VK_COLOR_SPACE_HDR10_HLG_EXT** specifies support for the HDR10 (BT2020 color space) to be
  displayed using the Hybrid Log Gamma (HLG) EOTF.

- **VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT** specifies support for the AdobeRGB color space to be
  displayed using a linear EOTF.

- **VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT** specifies support for the AdobeRGB color space to be
  displayed using the Gamma 2.2 EOTF.

- **VK_COLOR_SPACE_PASS_THROUGH_EXT** specifies that color components are used “as is”. This is
  intended to allow applications to supply data for color spaces not described here.

- **VK_COLOR_SPACE_DISPLAY_NATIVE_AMD** specifies support for the display’s native color space. This
  matches the color space expectations of AMD’s FreeSync2 standard, for displays supporting it.

**Note**

In the initial release of the **VK_KHR_surface** and **VK_KHR_swapchain** extensions, the token
**VKCOLORSPACE_SRGB_NONLINEAR_KHR** was used. Starting in the 2016-05-13
updates to the extension branches, matching release 1.0.13 of the core API
specification, **VK_COLOR_SPACE_SRGB_NONLINEAR_KHR** is used instead for consistency
with Vulkan naming rules. The older enum is still available for backwards
compatibility.

**Note**

In older versions of this extension **VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT** was
misnamed **VK_COLOR_SPACE_DCI_P3_LINEAR_EXT**. This has been updated to indicate
that it uses RGB color encoding, not XYZ. The old name is deprecated but is
maintained for backwards compatibility.

The color components of non-linear color space swap chain images **must** have had the appropriate
transfer function applied. The color space selected for the swap chain image will not affect the
processing of data written into the image by the implementation. Vulkan requires that all
implementations support the sRGB transfer function by use of an SRGB pixel format. Other transfer
functions, such as SMPTE 170M or SMPTE2084, **can** be performed by the application shader. This
extension defines enums for **VkColorSpaceKHR** that correspond to the following color spaces:
<table>
<thead>
<tr>
<th>Name</th>
<th>Red Primary</th>
<th>Green Primary</th>
<th>Blue Primary</th>
<th>White-point</th>
<th>Transfer function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCI-P3</td>
<td>1.000, 0.000</td>
<td>0.000, 1.000</td>
<td>0.000, 0.000</td>
<td>0.3333, 0.3333</td>
<td>DCI P3</td>
</tr>
<tr>
<td>Display-P3</td>
<td>0.680, 0.320</td>
<td>0.265, 0.690</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290</td>
<td>Display-P3</td>
</tr>
<tr>
<td>BT709</td>
<td>0.640, 0.330</td>
<td>0.300, 0.600</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290</td>
<td>ITU (SMPTE 170M)</td>
</tr>
<tr>
<td>sRGB</td>
<td>0.640, 0.330</td>
<td>0.300, 0.600</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290</td>
<td>sRGB</td>
</tr>
<tr>
<td>extended sRGB</td>
<td>0.640, 0.330</td>
<td>0.300, 0.600</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290</td>
<td>extended sRGB</td>
</tr>
<tr>
<td>HDR10_ST2084</td>
<td>0.708, 0.292</td>
<td>0.170, 0.797</td>
<td>0.131, 0.046</td>
<td>0.3127, 0.3290</td>
<td>ST2084 PQ</td>
</tr>
<tr>
<td>DOLBYVISION</td>
<td>0.708, 0.292</td>
<td>0.170, 0.797</td>
<td>0.131, 0.046</td>
<td>0.3127, 0.3290</td>
<td>ST2084 PQ</td>
</tr>
<tr>
<td>HDR10_HLG</td>
<td>0.708, 0.292</td>
<td>0.170, 0.797</td>
<td>0.131, 0.046</td>
<td>0.3127, 0.3290</td>
<td>HLG</td>
</tr>
<tr>
<td>AdobeRGB</td>
<td>0.640, 0.330</td>
<td>0.210, 0.710</td>
<td>0.150, 0.060</td>
<td>0.3127, 0.3290</td>
<td>AdobeRGB</td>
</tr>
</tbody>
</table>

The transfer functions are described in the “Transfer Functions” chapter of the Khronos Data Format Specification.

Except Display-P3 OETF, which is:

\[
E = \begin{cases} 
1.055 \times \frac{L^{\frac{1}{2.4}} - 0.055}{12.92} & \text{for } 0.0030186 \leq L \leq 1 \\
12.92 \times L & \text{for } 0 \leq L < 0.0030186 
\end{cases}
\]

where \( L \) is the linear value of a color channel and \( E \) is the encoded value (as stored in the image in memory).

**Note**

For most uses, the sRGB OETF is equivalent.

**33.5.3. Surface Presentation Mode Support**

To query the supported presentation modes for a surface, call:
// Provided by VK_KHR_surface
VkResult vkGetPhysicalDeviceSurfacePresentModesKHR(
    VkPhysicalDevice physicalDevice,
    VkSurfaceKHR surface,
    uint32_t* pPresentModeCount,
    VkPresentModeKHR* pPresentModes);

- **physicalDevice** is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- **surface** is the surface that will be associated with the swapchain.
- **pPresentModeCount** is a pointer to an integer related to the number of presentation modes available or queried, as described below.
- **pPresentModes** is either `NULL` or a pointer to an array of `VkPresentModeKHR` values, indicating the supported presentation modes.

If `pPresentModes` is `NULL`, then the number of presentation modes supported for the given `surface` is returned in `pPresentModeCount`. Otherwise, `pPresentModeCount` must point to a variable set by the user to the number of elements in the `pPresentModes` array, and on return the variable is overwritten with the number of values actually written to `pPresentModes`. If the value of `pPresentModeCount` is less than the number of presentation modes supported, at most `pPresentModeCount` values will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available modes were returned.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-physicalDevice-parameter
  **physicalDevice** must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-surface-parameter
  **surface** must be a valid `VkSurfaceKHR` handle

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-pPresentModeCount-parameter
  **pPresentModeCount** must be a valid pointer to a `uint32_t` value

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-pPresentModes-parameter
  If the value referenced by `pPresentModeCount` is not 0, and `pPresentModes` is not `NULL`, `pPresentModes` must be a valid pointer to an array of `pPresentModeCount` `VkPresentModeKHR` values

- VUID-vkGetPhysicalDeviceSurfacePresentModesKHR-commonparent
  Both of **physicalDevice**, and **surface** must have been created, allocated, or retrieved from the same `VkInstance`
Return Codes

Success

• VK_SUCCESS
• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_SURFACE_LOST_KHR

Alternatively, to query the supported presentation modes for a surface combined with select other fixed swapchain creation parameters, call:

```c
// Provided by VK_EXT_full_screen_exclusive
VkResult vkGetPhysicalDeviceSurfacePresentModes2EXT(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceSurfaceInfo2KHR* pSurfaceInfo,
    uint32_t* pPresentModeCount,
    VkPresentModeKHR* pPresentModes);
```

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `pSurfaceInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by `vkCreateSwapchainKHR`.
- `pPresentModeCount` is a pointer to an integer related to the number of presentation modes available or queried, as described below.
- `pPresentModes` is either `NULL` or a pointer to an array of `VkPresentModeKHR` values, indicating the supported presentation modes.

`vkGetPhysicalDeviceSurfacePresentModes2EXT` behaves similarly to `vkGetPhysicalDeviceSurfacePresentModesKHR`, with the ability to specify extended inputs via chained input structures.
Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceSurfacePresentModes2EXT-physicalDevice-parameter**
  physicalDevice must be a valid VkPhysicalDevice handle

- **VUID-vkGetPhysicalDeviceSurfacePresentModes2EXT-pSurfaceInfo-parameter**
  pSurfaceInfo must be a valid pointer to a valid VkPhysicalDeviceSurfaceInfo2KHR structure

- **VUID-vkGetPhysicalDeviceSurfacePresentModes2EXT-pPresentModeCount-parameter**
  pPresentModeCount must be a valid pointer to a uint32_t value

- **VUID-vkGetPhysicalDeviceSurfacePresentModes2EXT-pPresentModes-parameter**
  If the value referenced by pPresentModeCount is not 0, and pPresentModes is not NULL, pPresentModes must be a valid pointer to an array of pPresentModeCount VkPresentModeKHR values

Return Codes

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_SURFACE_LOST_KHR

Possible values of elements of the vkGetPhysicalDeviceSurfacePresentModesKHR::pPresentModes array, indicating the supported presentation modes for a surface, are:

```c
// Provided by VK_KHR_surface
typedef enum VkPresentModeKHR {
    VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
    VK_PRESENT_MODE_MAILBOX_KHR = 1,
    VK_PRESENT_MODE_FIFO_KHR = 2,
    VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3,
    // Provided by VK_KHR_shared_presentable_image
    VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR = 1000111000,
    // Provided by VK_KHR_shared_presentable_image
    VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR = 1000111001,
} VkPresentModeKHR;
```

- **VK_PRESENT_MODE_IMMEDIATE_KHR** specifies that the presentation engine does not wait for a vertical blanking period to update the current image, meaning this mode may result in visible tearing. No internal queuing of presentation requests is needed, as the requests are applied
immediately.

- **VK_PRESENT_MODE_MAILBOX_KHR** specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing *cannot* be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for re-use by the application. One request is removed from the queue and processed during each vertical blanking period in which the queue is non-empty.

- **VK_PRESENT_MODE_FIFO_KHR** specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing *cannot* be observed. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty. This is the only value of `presentMode` that is *required* to be supported.

- **VK_PRESENT_MODE_FIFO_RELAXED_KHR** specifies that the presentation engine generally waits for the next vertical blanking period to update the current image. If a vertical blanking period has already passed since the last update of the current image then the presentation engine does not wait for another vertical blanking period for the update, meaning this mode *may* result in visible tearing in this case. This mode is useful for reducing visual stutter with an application that will mostly present a new image before the next vertical blanking period, but may occasionally be late, and present a new image just after the next vertical blanking period. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during or after each vertical blanking period in which the queue is non-empty.

- **VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR** specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a *shared presentable image*. The presentation engine is only required to update the current image after a new presentation request is received. Therefore the application *must* make a presentation request whenever an update is required. However, the presentation engine *may* update the current image at any point, meaning this mode *may* result in visible tearing.

- **VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR** specifies that the presentation engine and application have concurrent access to a single image, which is referred to as a *shared presentable image*. The presentation engine periodically updates the current image on its regular refresh cycle. The application is only required to make one initial presentation request, after which the presentation engine *must* update the current image without any need for further presentation requests. The application *can* indicate the image contents have been updated by making a presentation request, but this does not guarantee the timing of when it will be updated. This mode *may* result in visible tearing if rendering to the image is not timed correctly.

The supported `VkImageUsageFlagBits` of the presentable images of a swapchain created for a surface *may* differ depending on the presentation mode, and can be determined as per the table below:

*Table 49. Presentable image usage queries*
For reference, the mode indicated by `VK_PRESENT_MODE_FIFO_KHR` is equivalent to the behavior of `{wgl|glX|egl}SwapBuffers` with a swap interval of 1, while the mode indicated by `VK_PRESENT_MODE_FIFO_RELAXED_KHR` is equivalent to the behavior of `{wgl|glX}SwapBuffers` with a swap interval of -1 (from the `{WGL|GLX}_EXT_swap_control_tear` extensions).

### 33.6. Full Screen Exclusive Control

Swapchains created with `fullScreenExclusive` set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT` must acquire and release exclusive full-screen access explicitly, using the following commands.

To acquire exclusive full-screen access for a swapchain, call:

```c
// Provided by VK_EXT_full_screen_exclusive
VkResult vkAcquireFullScreenExclusiveModeEXT(
    VkDevice device,
    VkSwapchainKHR swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to acquire exclusive full-screen access for.
A return value of VK_SUCCESS indicates that the swapchain successfully acquired exclusive full-screen access. The swapchain will retain this exclusivity until either the application releases exclusive full-screen access with vkReleaseFullScreenExclusiveModeEXT, destroys the swapchain, or if any of the swapchain commands return VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT indicating that the mode was lost because of platform-specific changes.

If the swapchain was unable to acquire exclusive full-screen access to the display then VK_ERROR_INITIALIZATION_FAILED is returned. An application can attempt to acquire exclusive full-screen access again for the same swapchain even if this command fails, or if VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT has been returned by a swapchain command.

Valid Usage (Implicit)

- VUID-vkAcquireFullScreenExclusiveModeEXT-device-parameter
device must be a valid VkDevice handle

- VUID-vkAcquireFullScreenExclusiveModeEXT-swapchain-parameter
swapchain must be a valid VkSwapchainKHR handle

- VUID-vkAcquireFullScreenExclusiveModeEXT-commonparent
Both of device, and swapchain must have been created, allocated, or retrieved from the same VkInstance

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED
- VK_ERROR_SURFACE_LOST_KHR
To release exclusive full-screen access from a swapchain, call:

```c
// Provided by VK_EXT_full_screen_exclusive
VkResult vkReleaseFullScreenExclusiveModeEXT(
    VkDevice device,
    VkSwapchainKHR swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to release exclusive full-screen access from.

**Note**

Applications will not be able to present to `swapchain` after this call until exclusive full-screen access is reacquired. This is usually useful to handle when an application is minimised or otherwise intends to stop presenting for a time.

### Valid Usage

- VUID-vkReleaseFullScreenExclusiveModeEXT-swapchain-02677  
  `swapchain` must not be in the retired state
- VUID-vkReleaseFullScreenExclusiveModeEXT-swapchain-02678  
  `swapchain` must be a swapchain created with a `VkSurfaceFullScreenExclusiveInfoEXT` structure, with `fullScreenExclusive` set to `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`

#### 33.7. Device Group Queries

A logical device that represents multiple physical devices may support presenting from images on more than one physical device, or combining images from multiple physical devices.

To query these capabilities, call:

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_surface
VkResult vkGetDeviceGroupPresentCapabilitiesKHR(
    VkDevice device,
    VkDeviceGroupPresentCapabilitiesKHR* pDeviceGroupPresentCapabilities);
```

- `device` is the logical device.
- `pDeviceGroupPresentCapabilities` is a pointer to a `VkDeviceGroupPresentCapabilitiesKHR` structure in which the device's capabilities are returned.
Valid Usage (Implicit)

- VUID-vkGetDeviceGroupPresentCapabilitiesKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetDeviceGroupPresentCapabilitiesKHR-pDeviceGroupPresentCapabilities-parameter
  pDeviceGroupPresentCapabilities must be a valid pointer to a VkDeviceGroupPresentCapabilitiesKHR structure

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDeviceGroupPresentCapabilitiesKHR structure is defined as:

```c
typedef struct VkDeviceGroupPresentCapabilitiesKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t presentMask[VK_MAX_DEVICE_GROUP_SIZE];
    VkDeviceGroupPresentModeFlagsKHR modes;
} VkDeviceGroupPresentCapabilitiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **presentMask** is an array of VK_MAX_DEVICE_GROUP_SIZE uint32_t masks, where the mask at element i is non-zero if physical device i has a presentation engine, and where bit j is set in element i if physical device i can present swapchain images from physical device j. If element i is non-zero, then bit i must be set.
- **modes** is a bitmask of VkDeviceGroupPresentModeFlagBitsKHR indicating which device group presentation modes are supported.

 modes always has VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR set.

The present mode flags are also used when presenting an image, in VkDeviceGroupPresentInfoKHR::mode.

If a device group only includes a single physical device, then **modes** must equal VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR.
Valid Usage (Implicit)

- **VUID-VkDeviceGroupPresentCapabilitiesKHR-sType-sType**
  
  sType **must** be **VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR**

- **VUID-VkDeviceGroupPresentCapabilitiesKHR-pNext-pNext**
  
  pNext **must** be **NULL**

Bits which **may** be set in **VkDeviceGroupPresentCapabilitiesKHR::modes** to indicate which device group presentation modes are supported are:

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_surface
typedef enum VkDeviceGroupPresentModeFlagBitsKHR {
    VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR = 0x00000001,
    VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR = 0x00000002,
    VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR = 0x00000004,
    VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR = 0x00000008,
} VkDeviceGroupPresentModeFlagBitsKHR;
```

- **VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR** specifies that any physical device with a presentation engine can present its own swapchain images.

- **VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR** specifies that any physical device with a presentation engine can present swapchain images from any physical device in its **presentMask**.

- **VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR** specifies that any physical device with a presentation engine can present the sum of swapchain images from any physical devices in its **presentMask**.

- **VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR** specifies that multiple physical devices with a presentation engine can each present their own swapchain images.

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_surface
typedef VkFlags VkDeviceGroupPresentModeFlagsKHR;
```

**VkDeviceGroupPresentModeFlagsKHR** is a bitmask type for setting a mask of zero or more **VkDeviceGroupPresentModeFlagBitsKHR**.

Some surfaces **may** not be capable of using all the device group present modes.

To query the supported device group present modes for a particular surface, call:
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with
// VK_KHR_surface
VkResult vkGetDeviceGroupSurfacePresentModesKHR(
    VkDevice                                    device,
    VkSurfaceKHR                                surface,
    VkDeviceGroupPresentModeFlagsKHR*           pModes);

• device is the logical device.
• surface is the surface.
• pModes is a pointer to a VkDeviceGroupPresentModeFlagsKHR in which the supported device
group present modes for the surface are returned.

The modes returned by this command are not invariant, and may change in response to the surface
being moved, resized, or occluded. These modes must be a subset of the modes returned by
vkGetDeviceGroupPresentCapabilitiesKHR.

Valid Usage (Implicit)

• VUID-vkGetDeviceGroupSurfacePresentModesKHR-device-parameter
device must be a valid VkDevice handle
• VUID-vkGetDeviceGroupSurfacePresentModesKHR-surface-parameter
surface must be a valid VkSurfaceKHR handle
• VUID-vkGetDeviceGroupSurfacePresentModesKHR-pModes-parameter
pModes must be a valid pointer to a VkDeviceGroupPresentModeFlagsKHR value
• VUID-vkGetDeviceGroupSurfacePresentModesKHR-commonparent
Both of device, and surface must have been created, allocated, or retrieved from the same
VkInstance

Host Synchronization

• Host access to surface must be externally synchronized

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_SURFACE_LOST_KHR
Alternatively, to query the supported device group presentation modes for a surface combined with select other fixed swapchain creation parameters, call:

```c
// Provided by VK_EXT_full_screen_exclusive with VK_KHR_device_group,
// VK_EXT_full_screen_exclusive with VK_VERSION_1_1
VkResult vkGetDeviceGroupSurfacePresentModes2EXT(
    VkDevice                                    device,
    const VkPhysicalDeviceSurfaceInfo2KHR*      pSurfaceInfo,
    VkDeviceGroupPresentModeFlagsKHR*           pModes);
```

- `device` is the logical device.
- `pSurfaceInfo` is a pointer to a `VkPhysicalDeviceSurfaceInfo2KHR` structure describing the surface and other fixed parameters that would be consumed by `vkCreateSwapchainKHR`.
- `pModes` is a pointer to a `VkDeviceGroupPresentModeFlagsKHR` in which the supported device group present modes for the surface are returned.

`vkGetDeviceGroupSurfacePresentModes2EXT` behaves similarly to `vkGetDeviceGroupSurfacePresentModesKHR`, with the ability to specify extended inputs via chained input structures.

**Valid Usage (Implicit)**

- VUID-vkGetDeviceGroupSurfacePresentModes2EXT-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkGetDeviceGroupSurfacePresentModes2EXT-pSurfaceInfo-parameter
  - `pSurfaceInfo` must be a valid pointer to a valid `VkPhysicalDeviceSurfaceInfo2KHR` structure
- VUID-vkGetDeviceGroupSurfacePresentModes2EXT-pModes-parameter
  - `pModes` must be a valid pointer to a `VkDeviceGroupPresentModeFlagsKHR` value

**Return Codes**

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_SURFACE_LOST_KHR`

When using `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, the application may need to know which regions of the surface are used when presenting locally on each physical device. Presentation of swapchain images to this surface need only have valid contents in the regions
returned by this command.

To query a set of rectangles used in presentation on the physical device, call:

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with
// VK_KHR_surface
VkResult vkGetPhysicalDevicePresentRectanglesKHR(
    VkPhysicalDevice                            physicalDevice,
    VkSurfaceKHR                                surface,
    uint32_t*                                   pRectCount,
    VkRect2D*                                   pRects);
```

- `physicalDevice` is the physical device.
- `surface` is the surface.
- `pRectCount` is a pointer to an integer related to the number of rectangles available or queried, as described below.
- `pRects` is either `NULL` or a pointer to an array of `VkRect2D` structures.

If `pRects` is `NULL`, then the number of rectangles used when presenting the given `surface` is returned in `pRectCount`. Otherwise, `pRectCount` must point to a variable set by the user to the number of elements in the `pRects` array, and on return the variable is overwritten with the number of structures actually written to `pRects`. If the value of `pRectCount` is less than the number of rectangles, at most `pRectCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available rectangles were returned.

The values returned by this command are not invariant, and may change in response to the surface being moved, resized, or occluded.

The rectangles returned by this command must not overlap.

**Valid Usage (Implicit)**

- `VUID-vkGetPhysicalDevicePresentRectanglesKHR-physicalDevice-parameter` `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `VUID-vkGetPhysicalDevicePresentRectanglesKHR-surface-parameter` `surface` must be a valid `VkSurfaceKHR` handle
- `VUID-vkGetPhysicalDevicePresentRectanglesKHR-pRectCount-parameter` `pRectCount` must be a valid pointer to a `uint32_t` value
- `VUID-vkGetPhysicalDevicePresentRectanglesKHR-pRects-parameter` If the value referenced by `pRectCount` is not 0, and `pRects` is not `NULL`, `pRects` must be a valid pointer to an array of `pRectCount` `VkRect2D` structures
- `VUID-vkGetPhysicalDevicePresentRectanglesKHR-commonparent` Both of `physicalDevice`, and `surface` must have been created, allocated, or retrieved from the same `VkInstance`
Host Synchronization

- Host access to surface must be externally synchronized

Return Codes

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

33.8. Display Timing Queries

Traditional game and real-time-animation applications frequently use VK_PRESENT_MODE_FIFO_KHR so that presentable images are updated during the vertical blanking period of a given refresh cycle (RC) of the presentation engine’s display. This avoids the visual anomaly known as tearing.

However, synchronizing the presentation of images with the RC does not prevent all forms of visual anomalies. Stuttering occurs when the geometry for each presentable image is not accurately positioned for when that image will be displayed. The geometry may appear to move too little some RCs, and too much for others. Sometimes the animation appears to freeze, when the same image is used for more than one RC.

In order to minimize stuttering, an application needs to correctly position their geometry for when the presentable image will be displayed to the user. To accomplish this, applications need various timing information about the presentation engine’s display. They need to know when presentable images were actually presented, and when they could have been presented. Applications also need to tell the presentation engine to display an image no sooner than a given time. This can allow the application’s animation to look smooth to the user, with no stuttering. The VK_GOOGLE_display_timing extension allows an application to satisfy these needs.

The presentation engine’s display typically refreshes the pixels that are displayed to the user on a periodic basis. The period may be fixed or variable. In many cases, the presentation engine is associated with fixed refresh rate (FRR) display technology, with a fixed refresh rate (RR, e.g. 60Hz). In some cases, the presentation engine is associated with variable refresh rate (VRR) display technology, where each refresh cycle (RC) can vary in length. This extension treats VRR displays as if they are FRR.

To query the duration of a refresh cycle (RC) for the presentation engine’s display, call:
// Provided by VK_GOOGLE_display_timing
VkResult vkGetRefreshCycleDurationGOOGLE(
    VkDevice device,
    VkSwapchainKHR swapchain,
    VkRefreshCycleDurationGOOGLE* pDisplayTimingProperties);

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to obtain the refresh duration for.
- `pDisplayTimingProperties` is a pointer to a `VkRefreshCycleDurationGOOGLE` structure.

### Valid Usage (Implicit)

- VUID-vkGetRefreshCycleDurationGOOGLE-device-parameter
  `device` must be a valid `VkDevice` handle
- VUID-vkGetRefreshCycleDurationGOOGLE-swapchain-parameter
  `swapchain` must be a valid `VkSwapchainKHR` handle
- VUID-vkGetRefreshCycleDurationGOOGLE-pDisplayTimingProperties-parameter
  `pDisplayTimingProperties` must be a valid pointer to a `VkRefreshCycleDurationGOOGLE` structure
- VUID-vkGetRefreshCycleDurationGOOGLE-commonparent
  Both of `device`, and `swapchain` must have been created, allocated, or retrieved from the same `VkInstance`

### Host Synchronization

- Host access to `swapchain` must be externally synchronized

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_SURFACE_LOST_KHR`

The `VkRefreshCycleDurationGOOGLE` structure is defined as:
• **refreshDuration** is the number of nanoseconds from the start of one refresh cycle to the next.

**Note**

The rate at which an application renders and presents new images is known as the image present rate (IPR, aka frame rate). The inverse of IPR, or the duration between each image present, is the image present duration (IPD). In order to provide a smooth, stutter-free animation, an application will want its IPD to be a multiple of **refreshDuration**. For example, if a display has a 60Hz refresh rate, **refreshDuration** will be a value in nanoseconds that is approximately equal to 16.67ms. In such a case, an application will want an IPD of 16.67ms (1X multiplier of **refreshDuration**), or 33.33ms (2X multiplier of **refreshDuration**), or 50.0ms (3X multiplier of **refreshDuration**), etc.

In order to determine a target IPD for a display (i.e. a multiple of **refreshDuration**), an application needs to determine when its images are actually displayed. Let’s say that an application has an initial target IPD of 16.67ms (1X multiplier of **refreshDuration**). It will therefore position the geometry of a new image 16.67ms later than the previous image. Let’s say that this application is running on slower hardware, so that it actually takes 20ms to render each new image. This will create visual anomalies, because the images will not be displayed to the user every 16.67ms, nor every 20ms. In this case, it is better for the application to adjust its target IPD to 33.33ms (i.e. a 2X multiplier of **refreshDuration**), and tell the presentation engine to not present images any sooner than every 33.33ms. This will allow the geometry to be correctly positioned for each presentable image.

Adjustments to an application’s IPD may be needed because different views of an application’s geometry can take different amounts of time to render. For example, looking at the sky may take less time to render than looking at multiple, complex items in a room. In general, it is good to not frequently change IPD, as that can cause visual anomalies. Adjustments to a larger IPD because of late images should happen quickly, but adjustments to a smaller IPD should only happen if the **actualPresentTime** and **earliestPresentTime** members of the **VkPastPresentationTimingGOOGLE** structure are consistently different, and if **presentMargin** is consistently large, over multiple images.

The implementation will maintain a limited amount of history of timing information about previous presents. Because of the asynchronous nature of the presentation engine, the timing information for a given vkQueuePresentKHR command will become available some time later. These time values can be asynchronously queried, and will be returned if available. All time values are in nanoseconds, relative to a monotonically-increasing clock (e.g. CLOCK_MONOTONIC (see clock_gettime(2)) on Android and Linux).
To asynchronously query the presentation engine, for newly-available timing information about one or more previous presents to a given swapchain, call:

```c
// Provided by VK_GOOGLE_display_timing
VkResult vkGetPastPresentationTimingGOOGLE(
    VkDevice                                    device,  // Provided by VK_GOOGLE_display_timing
    VkSwapchainKHR                              swapchain,
    uint32_t*                                   pPresentationTimingCount,  // Provided by VK_GOOGLE_display_timing
    VkPastPresentationTimingGOOGLE*             pPresentationTimings);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to obtain presentation timing information duration for.
- `pPresentationTimingCount` is a pointer to an integer related to the number of `VkPastPresentationTimingGOOGLE` structures to query, as described below.
- `pPresentationTimings` is either `NULL` or a pointer to an array of `VkPastPresentationTimingGOOGLE` structures.

If `pPresentationTimings` is `NULL`, then the number of newly-available timing records for the given swapchain is returned in `pPresentationTimingCount`. Otherwise, `pPresentationTimingCount` must point to a variable set by the user to the number of elements in the `pPresentationTimings` array, and on return the variable is overwritten with the number of structures actually written to `pPresentationTimings`. If the value of `pPresentationTimingCount` is less than the number of newly-available timing records, at most `pPresentationTimingCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available timing records were returned.

### Valid Usage (Implicit)

- **VUID-vkGetPastPresentationTimingGOOGLE-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkGetPastPresentationTimingGOOGLE-swapchain-parameter**
  - `swapchain` must be a valid `VkSwapchainKHR` handle
- **VUID-vkGetPastPresentationTimingGOOGLE-pPresentationTimingCount-parameter**
  - `pPresentationTimingCount` must be a valid pointer to a `uint32_t` value
- **VUID-vkGetPastPresentationTimingGOOGLE-pPresentationTimings-parameter**
  - If the value referenced by `pPresentationTimingCount` is not 0, and `pPresentationTimings` is not `NULL`, `pPresentationTimings` must be a valid pointer to an array of `pPresentationTimingCount` `VkPastPresentationTimingGOOGLE` structures
- **VUID-vkGetPastPresentationTimingGOOGLE-commonparent**
  - Both of `device`, and `swapchain` must have been created, allocated, or retrieved from the same `VkInstance`
Host Synchronization

- Host access to swapchain must be externally synchronized

Return Codes

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR
- VK_ERROR_SURFACE_LOST_KHR

The `VkPastPresentationTimingGOOGLE` structure is defined as:

```c
// Provided by VK_GOOGLE_display_timing
typedef struct VkPastPresentationTimingGOOGLE {
    uint32_t    presentID;
    uint64_t    desiredPresentTime;
    uint64_t    actualPresentTime;
    uint64_t    earliestPresentTime;
    uint64_t    presentMargin;
} VkPastPresentationTimingGOOGLE;
```

- `presentID` is an application-provided value that was given to a previous `vkQueuePresentKHR` command via `VkPresentTimeGOOGLE::presentID` (see below). It can be used to uniquely identify a previous present with the `vkQueuePresentKHR` command.

- `desiredPresentTime` is an application-provided value that was given to a previous `vkQueuePresentKHR` command via `VkPresentTimeGOOGLE::desiredPresentTime`. If non-zero, it was used by the application to indicate that an image not be presented any sooner than `desiredPresentTime`.

- `actualPresentTime` is the time when the image of the swapchain was actually displayed.

- `earliestPresentTime` is the time when the image of the swapchain could have been displayed. This may differ from `actualPresentTime` if the application requested that the image be presented no sooner than `VkPresentTimeGOOGLE::desiredPresentTime`.

- `presentMargin` is an indication of how early the `vkQueuePresentKHR` command was processed compared to how soon it needed to be processed, and still be presented at `earliestPresentTime`.

The results for a given swapchain and `presentID` are only returned once from
vkGetPastPresentationTimingGOOGLE.

The application can use the VkPastPresentationTimingGOOGLE values to occasionally adjust its timing. For example, if actualPresentTime is later than expected (e.g. one refreshDuration late), the application may increase its target IPD to a higher multiple of refreshDuration (e.g. decrease its frame rate from 60Hz to 30Hz). If actualPresentTime and earliestPresentTime are consistently different, and if presentMargin is consistently large enough, the application may decrease its target IPD to a smaller multiple of refreshDuration (e.g. increase its frame rate from 30Hz to 60Hz). If actualPresentTime and earliestPresentTime are same, and if presentMargin is consistently high, the application may delay the start of its input-render-present loop in order to decrease the latency between user input and the corresponding present (always leaving some margin in case a new image takes longer to render than the previous image). An application that desires its target IPD to always be the same as refreshDuration, can also adjust features until actualPresentTime is never late and presentMargin is satisfactory.

The full VK_GOOGLE_display_timing extension semantics are described for swapchains created with VK_PRESENT_MODE_FIFO_KHR. For example, non-zero values of VkPresentTimeGOOGLE::desiredPresentTime must be honored, and vkGetPastPresentationTimingGOOGLE should return a VkPastPresentationTimingGOOGLE structure with valid values for all images presented with vkQueuePresentKHR. The semantics for other present modes are as follows:

- **VK_PRESENT_MODE_IMMEDIATE_KHR.** The presentation engine may ignore non-zero values of VkPresentTimeGOOGLE::desiredPresentTime in favor of presenting immediately. The value of VkPastPresentationTimingGOOGLE::earliestPresentTime must be the same as VkPastPresentationTimingGOOGLE::actualPresentTime, which should be when the presentation engine displayed the image.

- **VK_PRESENT_MODE_MAILBOX_KHR.** The intention of using this present mode with this extension is to handle cases where an image is presented late, and the next image is presented soon enough to replace it at the next vertical blanking period. For images that are displayed to the user, the value of VkPastPresentationTimingGOOGLE::actualPresentTime must be when the image was displayed. For images that are not displayed to the user, vkGetPastPresentationTimingGOOGLE may not return a VkPastPresentationTimingGOOGLE structure, or it may return a VkPastPresentationTimingGOOGLE structure with the value of zero for both VkPastPresentationTimingGOOGLE::actualPresentTime and VkPastPresentationTimingGOOGLE::earliestPresentTime. It is possible that an application can submit images with VkPresentTimeGOOGLE::desiredPresentTime values such that new images may not be displayed. For example, if VkPresentTimeGOOGLE::desiredPresentTime is far enough in the future that an image is not presented before vkQueuePresentKHR is called to present another image, the first image will not be displayed to the user. If the application continues to do that, the presentation may not display new images.

- **VK_PRESENT_MODE_FIFO_RELAXED_KHR.** For images that are presented in time to be displayed at the next vertical blanking period, the semantics are identical as for VK_PRESENT_MODE_FIFO_RELAXED_KHR. For images that are presented late, and are displayed after the start of the vertical blanking period (i.e. with tearing), the values of VkPastPresentationTimingGOOGLE may be treated as if the image was displayed at the start of the vertical blanking period, or may be treated the same as for VK_PRESENT_MODE_IMMEDIATE_KHR.
33.9. WSI Swapchain

A swapchain object (a.k.a. swapchain) provides the ability to present rendering results to a surface. Swapchain objects are represented by `VkSwapchainKHR` handles:

```cpp
// Provided by VK_KHR_swapchain
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSwapchainKHR)
```

A swapchain is an abstraction for an array of presentable images that are associated with a surface. The presentable images are represented by `VkImage` objects created by the platform. One image (which can be an array image for multiview/stereoscopic-3D surfaces) is displayed at a time, but multiple images can be queued for presentation. An application renders to the image, and then queues the image for presentation to the surface.

A native window cannot be associated with more than one non-retired swapchain at a time. Further, swapchains cannot be created for native windows that have a non-Vulkan graphics API surface associated with them.

```
Note
The presentation engine is an abstraction for the platform's compositor or display engine.
```

```
The presentation engine may be synchronous or asynchronous with respect to the application and/or logical device.
```

```
Some implementations may use the device's graphics queue or dedicated presentation hardware to perform presentation.
```

The presentable images of a swapchain are owned by the presentation engine. An application can acquire use of a presentable image from the presentation engine. Use of a presentable image must occur only after the image is returned by `vkAcquireNextImageKHR`, and before it is released by `vkQueuePresentKHR`. This includes transitioning the image layout and rendering commands.

An application can acquire use of a presentable image with `vkAcquireNextImageKHR`. After acquiring a presentable image and before modifying it, the application must use a synchronization primitive to ensure that the presentation engine has finished reading from the image. The application can then transition the image's layout, queue rendering commands to it, etc. Finally, the application presents the image with `vkQueuePresentKHR`, which releases the acquisition of the image.

The presentation engine controls the order in which presentable images are acquired for use by the application.

```
Note
This allows the platform to handle situations which require out-of-order return of images after presentation. At the same time, it allows the application to generate command buffers referencing all of the images in the swapchain at initialization time, rather than in its main loop.
```

Chapter 33. Window System Integration (WSI) | 1819
How this all works is described below.

If a swapchain is created with `presentMode` set to either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, a single presentable image can be acquired, referred to as a shared presentable image. A shared presentable image may be concurrently accessed by the application and the presentation engine, without transitioning the image's layout after it is initially presented.

- With `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`, the presentation engine is only required to update to the latest contents of a shared presentable image after a present. The application must call `vkQueuePresentKHR` to guarantee an update. However, the presentation engine may update from it at any time.

- With `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, the presentation engine will automatically present the latest contents of a shared presentable image during every refresh cycle. The application is only required to make one initial call to `vkQueuePresentKHR`, after which the presentation engine will update from it without any need for further present calls. The application can indicate the image contents have been updated by calling `vkQueuePresentKHR`, but this does not guarantee the timing of when updates will occur.

The presentation engine may access a shared presentable image at any time after it is first presented. To avoid tearing, an application should coordinate access with the presentation engine. This requires presentation engine timing information through platform-specific mechanisms and ensuring that color attachment writes are made available during the portion of the presentation engine's refresh cycle they are intended for.

```
// Provided by VK_KHR_shared_presentable_image
VkResult vkGetSwapchainStatusKHR(
    VkDevice                                    device,
    VkSwapchainKHR                              swapchain);
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to query.

Note
The `VK_KHR_shared_presentable_image` extension does not provide functionality for determining the timing of the presentation engine's refresh cycles.
Valid Usage (Implicit)

- **VUID-vkGetSwapchainStatusKHR-device-parameter**
  
  **device** must be a valid `VkDevice` handle

- **VUID-vkGetSwapchainStatusKHR-swapchain-parameter**
  
  **swapchain** must be a valid `VkSwapchainKHR` handle

- **VUID-vkGetSwapchainStatusKHR-commonparent**
  
  Both of **device**, and **swapchain** must have been created, allocated, or retrieved from the same `VkInstance`

Host Synchronization

- Host access to **swapchain** must be externally synchronized

Return Codes

**Success**

- `VK_SUCCESS`
- `VK_SUBOPTIMAL_KHR`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_OUT_OF_DATE_KHR`
- `VK_ERROR_SURFACE_LOST_KHR`
- `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT`

The possible return values for `vkGetSwapchainStatusKHR` should be interpreted as follows:

- **VK_SUCCESS** specifies the presentation engine is presenting the contents of the shared presentable image, as per the swapchain’s `VkPresentModeKHR`.

- **VK_SUBOPTIMAL_KHR** the swapchain no longer matches the surface properties exactly, but the presentation engine is presenting the contents of the shared presentable image, as per the swapchain’s `VkPresentModeKHR`.

- **VK_ERROR_OUT_OF_DATE_KHR** the surface has changed in such a way that it is no longer compatible with the swapchain.

- **VK_ERROR_SURFACE_LOST_KHR** the surface is no longer available.
Note
The swapchain state may be cached by implementations, so applications should regularly call `vkGetSwapchainStatusKHR` when using a swapchain with `VkPresentModeKHR` set to `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`. To create a swapchain, call:

```cpp
// Provided by VK_KHR_swapchain
VkResult vkCreateSwapchainKHR(
    VkDevice device, const VkSwapchainCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks* pAllocator, VkSwapchainKHR* pSwapchain);
```

- `device` is the device to create the swapchain for.
- `pCreateInfo` is a pointer to a `VkSwapchainCreateInfoKHR` structure specifying the parameters of the created swapchain.
- `pAllocator` is the allocator used for host memory allocated for the swapchain object when there is no more specific allocator available (see Memory Allocation).
- `pSwapchain` is a pointer to a `VkSwapchainKHR` handle in which the created swapchain object will be returned.

If the `oldSwapchain` parameter of `pCreateInfo` is a valid swapchain, which has exclusive full-screen access, that access is released from `oldSwapchain`. If the command succeeds in this case, the newly created swapchain will automatically acquire exclusive full-screen access from `oldSwapchain`.

Note
This implicit transfer is intended to avoid exiting and entering full-screen exclusive mode, which may otherwise cause unwanted visual updates to the display.

In some cases, swapchain creation may fail if exclusive full-screen mode is requested for application control, but for some implementation-specific reason exclusive full-screen access is unavailable for the particular combination of parameters provided. If this occurs, `VK_ERROR_INITIALIZATION_FAILED` will be returned.

Note
In particular, it will fail if the `imageExtent` member of `pCreateInfo` does not match the extents of the monitor. Other reasons for failure may include the app not being set as high-dpi aware, or if the physical device and monitor are not compatible in this mode.

When the `VkSurfaceKHR` in `VkSwapchainCreateInfoKHR` is a display surface, then the `VkDisplayModeKHR` in display surface’s `VkDisplaySurfaceCreateInfoKHR` is associated with a particular `VkDisplayKHR`. Swapchain creation may fail if that `VkDisplayKHR` is not acquired by the
application. In this scenario `VK_ERROR_INITIALIZATION_FAILED` is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateSwapchainKHR-device-parameter**
  device must be a valid `VkDevice` handle

- **VUID-vkCreateSwapchainKHR-pCreateInfo-parameter**
  pCreateInfo must be a valid pointer to a valid `VkSwapchainCreateInfoKHR` structure

- **VUID-vkCreateSwapchainKHR-pAllocator-parameter**
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid `VkAllocationCallbacks` structure

- **VUID-vkCreateSwapchainKHR-pSwapchain-parameter**
  pSwapchain must be a valid pointer to a `VkSwapchainKHR` handle

### Host Synchronization

- Host access to pCreateInfo->surface must be externally synchronized
- Host access to pCreateInfo->oldSwapchain must be externally synchronized

### Return Codes

#### Success

- `VK_SUCCESS`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`
- `VK_ERROR_DEVICE_LOST`
- `VK_ERROR_SURFACE_LOST_KHR`
- `VK_ERROR_NATIVE_WINDOW_IN_USE_KHR`
- `VK_ERROR_INITIALIZATION_FAILED`

The `VkSwapchainCreateInfoKHR` structure is defined as:
// Provided by VK_KHR_swapchain

typedef struct VkSwapchainCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkSwapchainCreateFlagsKHR flags;
    VkSurfaceKHR surface;
    uint32_t minImageCount;
    VkFormat imageFormat;
    VkColorSpaceKHR imageColorSpace;
    VkExtent2D imageExtent;
    uint32_t imageArrayLayers;
    VkImageUsageFlags imageUsage;
    VkSharingMode imageSharingMode;
    uint32_t queueFamilyIndexCount;
    const uint32_t* pQueueFamilyIndices;
    VkSurfaceTransformFlagBitsKHR preTransform;
    VkCompositeAlphaFlagBitsKHR compositeAlpha;
    VkPresentModeKHR presentMode;
    VkBool32 clipped;
    VkSwapchainKHR oldSwapchain;
} VkSwapchainCreateInfoKHR;

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **flags** is a bitmask of **VkSwapchainCreateFlagBitsKHR** indicating parameters of the swapchain creation.

• **surface** is the surface onto which the swapchain will present images. If the creation succeeds, the swapchain becomes associated with **surface**.

• **minImageCount** is the minimum number of presentable images that the application needs. The implementation will either create the swapchain with at least that many images, or it will fail to create the swapchain.

• **imageFormat** is a **VkFormat** value specifying the format the swapchain image(s) will be created with.

• **imageColorSpace** is a **VkColorSpaceKHR** value specifying the way the swapchain interprets image data.

• **imageExtent** is the size (in pixels) of the swapchain image(s). The behavior is platform-dependent if the image extent does not match the surface's **currentExtent** as returned by **vkGetPhysicalDeviceSurfaceCapabilitiesKHR**.

---

**Note**

On some platforms, it is normal that **maxImageExtent may** become *(0, 0)*, for example when the window is minimized. In such a case, it is not possible to create a swapchain due to the Valid Usage requirements.

• **imageArrayLayers** is the number of views in a multiview/stereo surface. For non-stereoscopic-3D
Applications, this value is 1.

- **imageUsage** is a bitmask of `VkImageUsageFlagBits` describing the intended usage of the (acquired) swapchain images.
- **imageSharingMode** is the sharing mode used for the image(s) of the swapchain.
- **queueFamilyIndexCount** is the number of queue families having access to the image(s) of the swapchain when `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`.
- **pNextFamilyIndices** is a pointer to an array of queue family indices having access to the image(s) of the swapchain when `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`.
- **preTransform** is a `VkSurfaceTransformFlagBitsKHR` value describing the transform, relative to the presentation engine's natural orientation, applied to the image content prior to presentation. If it does not match the `currentTransform` value returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`, the presentation engine will transform the image content as part of the presentation operation.
- **compositeAlpha** is a `VkCompositeAlphaFlagBitsKHR` value indicating the alpha compositing mode to use when this surface is composited together with other surfaces on certain window systems.
- **presentMode** is the presentation mode the swapchain will use. A swapchain's present mode determines how incoming present requests will be processed and queued internally.
- **clipped** specifies whether the Vulkan implementation is allowed to discard rendering operations that affect regions of the surface that are not visible.
  - If set to `VK_TRUE`, the presentable images associated with the swapchain may not own all of their pixels. Pixels in the presentable images that correspond to regions of the target surface obscured by another window on the desktop, or subject to some other clipping mechanism will have undefined content when read back. Fragment shaders may not execute for these pixels, and thus any side effects they would have had will not occur. Setting `VK_TRUE` does not guarantee any clipping will occur, but allows more efficient presentation methods to be used on some platforms.
  - If set to `VK_FALSE`, presentable images associated with the swapchain will own all of the pixels they contain.

**Note**

Applications **should** set this value to `VK_TRUE` if they do not expect to read back the content of presentable images before presenting them or after reacquiring them, and if their fragment shaders do not have any side effects that require them to run for all pixels in the presentable image.

- **oldSwapchain** is `VK_NULL_HANDLE`, or the existing non-retired swapchain currently associated with `surface`. Providing a valid `oldSwapchain` may aid in the resource reuse, and also allows the application to still present any images that are already acquired from it.

Upon calling `vkCreateSwapchainKHR` with an `oldSwapchain` that is not `VK_NULL_HANDLE`, `oldSwapchain` is retired — even if creation of the new swapchain fails. The new swapchain is created in the non-retired state whether or not `oldSwapchain` is `VK_NULL_HANDLE`.

Upon calling `vkCreateSwapchainKHR` with an `oldSwapchain` that is not `VK_NULL_HANDLE`, any images
from oldSwapchain that are not acquired by the application may be freed by the implementation, which may occur even if creation of the new swapchain fails. The application can destroy oldSwapchain to free all memory associated with oldSwapchain.

**Note**

Multiple retired swapchains can be associated with the same VkSurfaceKHR through multiple uses of oldSwapchain that outnumber calls to vkDestroySwapchainKHR.

After oldSwapchain is retired, the application can pass to vkQueuePresentKHR any images it had already acquired from oldSwapchain. E.g., an application may present an image from the old swapchain before an image from the new swapchain is ready to be presented. As usual, vkQueuePresentKHR may fail if oldSwapchain has entered a state that causes VK_ERROR_OUT_OF_DATE_KHR to be returned.

The application can continue to use a shared presentable image obtained from oldSwapchain until a presentable image is acquired from the new swapchain, as long as it has not entered a state that causes it to return VK_ERROR_OUT_OF_DATE_KHR.
Valid Usage

- **VUID-VkSwapchainCreateInfoKHR-surface-01270**
  The `surface` must be a surface that is supported by the device as determined using `vkGetPhysicalDeviceSurfaceSupportKHR`.

- **VUID-VkSwapchainCreateInfoKHR-minImageCount-01272**
  The `minImageCount` must be less than or equal to the value returned in the `maxImageCount` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface if the returned `maxImageCount` is not zero.

- **VUID-VkSwapchainCreateInfoKHR-presentMode-02839**
  If `presentMode` is not `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` nor `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, then `minImageCount` must be greater than or equal to the value returned in the `minImageCount` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface.

- **VUID-VkSwapchainCreateInfoKHR-minImageCount-01383**
  The `minImageCount` must be 1 if `presentMode` is either `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`.

- **VUID-VkSwapchainCreateInfoKHR-imageFormat-01273**
  The `imageFormat` and `imageColorSpace` must match the format and colorSpace members, respectively, of one of the `VkSurfaceFormatKHR` structures returned by `vkGetPhysicalDeviceSurfaceFormatsKHR` for the surface.

- **VUID-VkSwapchainCreateInfoKHR-imageExtent-01274**
  The `imageExtent` must be between `minImageExtent` and `maxImageExtent`, inclusive, where `minImageExtent` and `maxImageExtent` are members of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface.

- **VUID-VkSwapchainCreateInfoKHR-imageExtent-01689**
  The `imageExtent` members `width` and `height` must both be non-zero.

- **VUID-VkSwapchainCreateInfoKHR-imageArrayLayers-01275**
  The `imageArrayLayers` must be greater than 0 and less than or equal to the `maxImageArrayLayers` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface.

- **VUID-VkSwapchainCreateInfoKHR-presentMode-01427**
  If `presentMode` is `VK_PRESENT_MODE_IMMEDIATE_KHR`, `VK_PRESENT_MODE_MAILBOX_KHR`, `VK_PRESENT_MODE_FIFO_KHR` or `VK_PRESENT_MODE_FIFO_RELAXED_KHR`, `imageUsage` must be a subset of the supported usage flags present in the `supportedUsageFlags` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface.

- **VUID-VkSwapchainCreateInfoKHR-imageUsage-01384**
  If `presentMode` is `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR` or `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`, `imageUsage` must be a subset of the supported usage flags present in the `sharedPresentSupportedUsageFlags` member of the
VkSharedPresentSurfaceCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilities2KHR for surface

- VUID-VkSwapchainCreateInfoKHR-imageSharingMode-01277
  If `imageSharingMode` is VK_SHARING_MODE_CONCURRENT, `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount` `uint32_t` values

- VUID-VkSwapchainCreateInfoKHR-imageSharingMode-01278
  If `imageSharingMode` is VK_SHARING_MODE_CONCURRENT, `queueFamilyIndexCount` must be greater than 1

- VUID-VkSwapchainCreateInfoKHR-imageSharingMode-01428
  If `imageSharingMode` is VK_SHARING_MODE_CONCURRENT, each element of `pQueueFamilyIndices` must be unique and must be less than `queueFamilyPropertyCount` returned by either `vkGetPhysicalDeviceQueueFamilyProperties` or `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device

- VUID-VkSwapchainCreateInfoKHR-preTransform-01279
  `preTransform` must be one of the bits present in the `supportedTransforms` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface

- VUID-VkSwapchainCreateInfoKHR-compositeAlpha-01280
  `compositeAlpha` must be one of the bits present in the `supportedCompositeAlpha` member of the `VkSurfaceCapabilitiesKHR` structure returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` for the surface

- VUID-VkSwapchainCreateInfoKHR-presentMode-01281
  `presentMode` must be one of the `VkPresentModeKHR` values returned by `vkGetPhysicalDeviceSurfacePresentModesKHR` for the surface

- VUID-VkSwapchainCreateInfoKHR-physicalDeviceCount-01429
  If the logical device was created with `VkDeviceGroupDeviceCreateInfo`::`physicalDeviceCount` equal to 1, `flags` must not contain `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`

- VUID-VkSwapchainCreateInfoKHR-oldSwapchain-01933
  If `oldSwapchain` is not VK_NULL_HANDLE, `oldSwapchain` must be a non-retired swapchain associated with native window referred to by `surface`

- VUID-VkSwapchainCreateInfoKHR-imageFormat-01778
  The implied image creation parameters of the swapchain must be supported as reported by `vkGetPhysicalDeviceImageFormatProperties`

- VUID-VkSwapchainCreateInfoKHR-flags-03168
  If `flags` contains `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR` then the `pNext` chain must include a `VkImageFormatListCreateInfo` structure with a `viewFormatCount` greater than zero and `pViewFormats` must have an element equal to `imageFormat`

- VUID-VkSwapchainCreateInfoKHR-pNext-04099
  If a `VkImageFormatListCreateInfo` structure was included in the `pNext` chain and `VkImageFormatListCreateInfo`::`viewFormatCount` is not zero then all of the formats in `VkImageFormatListCreateInfo`::`pViewFormats` must be compatible with the `format` as described in the compatibility table
If flags does not contain VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR and the pNext chain include a VkImageFormatListCreateInfo structure then VkImageFormatListCreateInfo::viewFormatCount must be 0 or 1.

If flags contains VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR, then VkSurfaceProtectedCapabilitiesKHR::supportsProtected must be VK_TRUE in the VkSurfaceProtectedCapabilitiesKHR structure returned by vkGetPhysicalDeviceSurfaceCapabilities2KHR for surface.

If the pNext chain includes a VkSurfaceFullScreenExclusiveInfoEXT structure with its fullscreenExclusive member set to VKFULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT, and surface was created using vkCreateWin32SurfaceKHR, a VkSurfaceFullScreenExclusiveWin32InfoEXT structure must be included in the pNext chain.
Valid Usage (Implicit)

- **VUID-VkSwapchainCreateInfoKHR-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR**

- **VUID-VkSwapchainCreateInfoKHR-pNext-pNext**
  
  Each **pNext** member of any structure (including this one) in the **pNext** chain must be either **NULL** or a pointer to a valid instance of **VkDeviceGroupSwapchainCreateInfoKHR**, **VkImageFormatListCreateInfo**, **VkSurfaceFullScreenExclusiveInfoEXT**, **VkSurfaceFullScreenExclusiveWin32InfoEXT**, **VkSwapchainCounterCreateInfoEXT**, or **VkSwapchainDisplayNativeHdrCreateInfoAMD**

- **VUID-VkSwapchainCreateInfoKHR-sType-unique**
  
  The **sType** value of each struct in the **pNext** chain must be unique

- **VUID-VkSwapchainCreateInfoKHR-flags-parameter**
  
  **flags** must be a valid combination of **VkSwapchainCreateFlagBitsKHR** values

- **VUID-VkSwapchainCreateInfoKHR-surface-parameter**
  
  **surface** must be a valid **VkSurfaceKHR** handle

- **VUID-VkSwapchainCreateInfoKHR-imageFormat-parameter**
  
  **imageFormat** must be a valid **VkFormat** value

- **VUID-VkSwapchainCreateInfoKHR-imageColorSpace-parameter**
  
  **imageColorSpace** must be a valid **VkColorSpaceKHR** value

- **VUID-VkSwapchainCreateInfoKHR-imageUsage-parameter**
  
  **imageUsage** must be a valid combination of **VkImageUsageFlagBits** values

- **VUID-VkSwapchainCreateInfoKHR-imageUsage-requiredbitmask**
  
  **imageUsage** must not be 0

- **VUID-VkSwapchainCreateInfoKHR-imageSharingMode-parameter**
  
  **imageSharingMode** must be a valid **VkSharingMode** value

- **VUID-VkSwapchainCreateInfoKHR-preTransform-parameter**
  
  **preTransform** must be a valid **VkSurfaceTransformFlagBitsKHR** value

- **VUID-VkSwapchainCreateInfoKHR-compositeAlpha-parameter**
  
  **compositeAlpha** must be a valid **VkCompositeAlphaFlagBitsKHR** value

- **VUID-VkSwapchainCreateInfoKHR-presentMode-parameter**
  
  **presentMode** must be a valid **VkPresentModeKHR** value

- **VUID-VkSwapchainCreateInfoKHR-oldSwapchain-parameter**
  
  If **oldSwapchain** is not **VK_NULL_HANDLE**, **oldSwapchain** must be a valid **VkSwapchainKHR** handle

- **VUID-VkSwapchainCreateInfoKHR-oldSwapchain-parent**
  
  If **oldSwapchain** is a valid handle, it must have been created, allocated, or retrieved from **surface**

- **VUID-VkSwapchainCreateInfoKHR-commonparent**
  
  Both of **oldSwapchain**, and **surface** that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same **VkInstance**
Bits which can be set in `VkSwapchainCreateInfoKHR::flags`, specifying parameters of swapchain creation, are:

```c
// Provided by VK_KHR_swapchain
typedef enum VkSwapchainCreateFlagBitsKHR {
    // Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with
    // VK_KHR_swapchain
    VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR = 0x00000001,
    // Provided by VK_KHR_swapchain with VK_VERSION_1_1
    VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR = 0x00000002,
    // Provided by VK_KHR_swapchain_mutable_format
    VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR = 0x00000004,
} VkSwapchainCreateFlagBitsKHR;
```

- `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR` specifies that images created from the swapchain (i.e. with the `swapchain` member of `VkImageSwapchainCreateInfoKHR` set to this swapchain's handle) must use `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT`.
- `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR` specifies that images created from the swapchain are protected images.
- `VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR` specifies that the images of the swapchain can be used to create a `VkImageView` with a different format than what the swapchain was created with. The list of allowed image view formats is specified by adding a `VkImageFormatListCreateInfo` structure to the `pNext` chain of `VkSwapchainCreateInfoKHR`. In addition, this flag also specifies that the swapchain can be created with usage flags that are not supported for the format the swapchain is created with but are supported for at least one of the allowed image view formats.

```c
// Provided by VK_KHR_swapchain
typedef VkFlags VkSwapchainCreateFlagsKHR;
```

`VkSwapchainCreateFlagsKHR` is a bitmask type for setting a mask of zero or more `VkSwapchainCreateFlagBitsKHR`.

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkDeviceGroupSwapchainCreateInfoKHR` structure, then that structure includes a set of device group present modes that the swapchain can be used with.

The `VkDeviceGroupSwapchainCreateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with
// VK_KHR_swapchain
typedef struct VkDeviceGroupSwapchainCreateInfoKHR {
    VkStructureType                     sType;
    const void*                         pNext;
    VkDeviceGroupPresentModeFlagsKHR    modes;
} VkDeviceGroupSwapchainCreateInfoKHR;
```
• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **modes** is a bitfield of modes that the swapchain can be used with.

If this structure is not present, **modes** is considered to be `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`.

### Valid Usage (Implicit)

- VUID-VkDeviceGroupSwapchainCreateInfoKHR-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`

- VUID-VkDeviceGroupSwapchainCreateInfoKHR-modes-parameter
  
  **modes** must be a valid combination of `VkDeviceGroupPresentModeFlagBitsKHR` values

- VUID-VkDeviceGroupSwapchainCreateInfoKHR-modes-requiredbitmask
  
  **modes** must not be 0

If the **pNext** chain of **VkSwapchainCreateInfoKHR** includes a `VkSwapchainDisplayNativeHdrCreateInfoAMD` structure, then that structure includes additional swapchain creation parameters specific to display native HDR support.

The `VkSwapchainDisplayNativeHdrCreateInfoAMD` structure is defined as:

```c
// Provided by VK_AMD_display_native_hdr
typedef struct VkSwapchainDisplayNativeHdrCreateInfoAMD {
    VkStructureType sType;
    const void*        pNext;
    VkBool32           localDimmingEnable;
} VkSwapchainDisplayNativeHdrCreateInfoAMD;
```

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **localDimmingEnable** specifies whether local dimming is enabled for the swapchain.

If the **pNext** chain of **VkSwapchainCreateInfoKHR** does not include this structure, the default value for **localDimmingEnable** is **VK_TRUE**, meaning local dimming is initially enabled for the swapchain.

### Valid Usage (Implicit)

- VUID-VkSwapchainDisplayNativeHdrCreateInfoAMD-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_SWAPCHAIN_DISPLAY_NATIVE_HDR_CREATE_INFO_AMD`
It is only valid to set `localDimmingEnable` to `VK_TRUE` if `VkDisplayNativeHdrSurfaceCapabilitiesAMD::localDimmingSupport` is supported.

The local dimming HDR setting may also be changed over the life of a swapchain by calling:

```c
// Provided by VK_AMD_display_native_hdr
void vkSetLocalDimmingAMD(
    VkDevice device,
    VkSwapchainKHR swapChain,
    VkBool32 localDimmingEnable);
```

- `device` is the device associated with `swapChain`.
- `swapChain` handle to enable local dimming.
- `localDimmingEnable` specifies whether local dimming is enabled for the swapchain.

If the `pNext` chain of `VkSwapchainCreateInfoKHR` includes a `VkSurfaceFullScreenExclusiveInfoEXT` structure, then that structure specifies the application's preferred full-screen presentation behavior. If this structure is not present, `fullScreenExclusive` is considered to be `VK_FULL_SCREEN_EXCLUSIVE_DEFAULT_EXT`.

To enable surface counters when creating a swapchain, add a `VkSwapchainCounterCreateInfoEXT` structure to the `pNext` chain of `VkSwapchainCreateInfoKHR`. `VkSwapchainCounterCreateInfoEXT` is defined as:
typedef struct VkSwapchainCounterCreateInfoEXT {
    VkStructureType             sType;
    const void*                 pNext;
    VkSurfaceCounterFlagsEXT    surfaceCounters;
} VkSwapchainCounterCreateInfoEXT;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `surfaceCounters` is a bitmask of `VkSurfaceCounterFlagBitsEXT` specifying surface counters to enable for the swapchain.

**Valid Usage**

- VUID-VkSwapchainCounterCreateInfoEXT-surfaceCounters-01244
  The bits in `surfaceCounters` must be supported by `VkSwapchainCreateInfoKHR::surface`, as reported by `vkGetPhysicalDeviceSurfaceCapabilities2EXT`

**Valid Usage (Implicit)**

- VUID-VkSwapchainCounterCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_SWAPCHAIN_COUNTER_CREATE_INFO_EXT`
- VUID-VkSwapchainCounterCreateInfoEXT-surfaceCounters-parameter
  `surfaceCounters` must be a valid combination of `VkSurfaceCounterFlagBitsEXT` values

The requested counters become active when the first presentation command for the associated swapchain is processed by the presentation engine. To query the value of an active counter, use:

```c
// Provided by VK_EXT_display_control
VkResult vkGetSwapchainCounterEXT(
    VkDevice                                    device,
    VkSwapchainKHR                              swapchain,
    VkSurfaceCounterFlagBitsEXT                 counter,
    uint64_t*                                   pCounterValue);
```

- `device` is the `VkDevice` associated with `swapchain`.
- `swapchain` is the swapchain from which to query the counter value.
- `counter` is the counter to query.
- `pCounterValue` will return the current value of the counter.

If a counter is not available because the swapchain is out of date, the implementation may return `VK_ERROR_OUT_OF_DATE_KHR`.
Valid Usage

- VUID-vkGetSwapchainCounterEXT-swapchain-01245
  One or more present commands on swapchain must have been processed by the presentation engine

Valid Usage (Implicit)

- VUID-vkGetSwapchainCounterEXT-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetSwapchainCounterEXT-swapchain-parameter
  swapchain must be a valid VkSwapchainKHR handle

- VUID-vkGetSwapchainCounterEXT-counter-parameter
  counter must be a valid VkSurfaceCounterFlagBitsEXT value

- VUID-vkGetSwapchainCounterEXT-pCounterValue-parameter
  pCounterValue must be a valid pointer to a uint64_t value

- VUID-vkGetSwapchainCounterEXT-commonparent
  Both of device, and swapchain must have been created, allocated, or retrieved from the same VkInstance

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR

As mentioned above, if vkCreateSwapchainKHR succeeds, it will return a handle to a swapchain containing an array of at least minImageCount presentable images.

While acquired by the application, presentable images can be used in any way that equivalent non-presentable images can be used. A presentable image is equivalent to a non-presentable image created with the following VkImageCreateInfo parameters:
<table>
<thead>
<tr>
<th>VkImageCreateInfo Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT is set if VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR is set</td>
</tr>
<tr>
<td></td>
<td>VK_IMAGE_CREATE_PROTECTED_BIT is set if VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR is set</td>
</tr>
<tr>
<td></td>
<td>VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT and VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR are both set if VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR is set</td>
</tr>
<tr>
<td></td>
<td>all other bits are unset</td>
</tr>
<tr>
<td>imageType</td>
<td>VK_IMAGE_TYPE_2D</td>
</tr>
<tr>
<td>format</td>
<td>pCreateInfo-&gt;imageFormat</td>
</tr>
<tr>
<td>extent</td>
<td>{pCreateInfo-&gt;imageExtent.width, pCreateInfo-&gt;imageExtent.height, 1}</td>
</tr>
<tr>
<td>mipLevels</td>
<td>1</td>
</tr>
<tr>
<td>arrayLayers</td>
<td>pCreateInfo-&gt;imageArrayLayers</td>
</tr>
<tr>
<td>samples</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
</tr>
<tr>
<td>tiling</td>
<td>VK_IMAGE_TILING_OPTIMAL</td>
</tr>
<tr>
<td>usage</td>
<td>pCreateInfo-&gt;imageUsage</td>
</tr>
<tr>
<td>sharingMode</td>
<td>pCreateInfo-&gt;imageSharingMode</td>
</tr>
<tr>
<td>queueFamilyIndexCount</td>
<td>pCreateInfo-&gt;queueFamilyIndexCount</td>
</tr>
<tr>
<td>pQueueFamilyIndices</td>
<td>pCreateInfo-&gt;pQueueFamilyIndices</td>
</tr>
<tr>
<td>initialLayout</td>
<td>VK_IMAGE_LAYOUT_UNDEFINED</td>
</tr>
</tbody>
</table>

The surface must not be destroyed until after the swapchain is destroyed.

If oldSwapchain is VK_NULL_HANDLE, and the native window referred to by surface is already associated with a Vulkan swapchain, VK_ERROR_NATIVE_WINDOW_IN_USE_KHR must be returned.

If the native window referred to by surface is already associated with a non-Vulkan graphics API surface, VK_ERROR_NATIVE_WINDOW_IN_USE_KHR must be returned.

The native window referred to by surface must not become associated with a non-Vulkan graphics API surface before all associated Vulkan swapchains have been destroyed.

Like core functions, several WSI functions, including vkCreateSwapchainKHR return VK_ERROR_DEVICE_LOST if the logical device was lost. See Lost Device. As with most core objects, VkSwapchainKHR is a child of the device and is affected by the lost state; it must be destroyed before destroying the VkDevice. However, VkSurfaceKHR is not a child of any VkDevice and is not otherwise affected by the lost device. After successfully recreating a VkDevice, the same VkSurfaceKHR can be
used to create a new `VkSwapchainKHR`, provided the previous one was destroyed.

**Note**

As mentioned in Lost Device, after a lost device event, the `VkPhysicalDevice` **may** also be lost. If other `VkPhysicalDevice` are available, they **can** be used together with the same `VkSurfaceKHR` to create the new `VkSwapchainKHR`, however the application **must** query the surface capabilities again, because they **may** differ on a per-physical device basis.

To destroy a swapchain object call:

```c
// Provided by VK_KHR_swapchain
void vkDestroySwapchainKHR(
    VkDevice device,
    VkSwapchainKHR swapchain,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the `VkDevice` associated with `swapchain`.
- **swapchain** is the swapchain to destroy.
- **pAllocator** is the allocator used for host memory allocated for the swapchain object when there is no more specific allocator available (see Memory Allocation).

The application **must** not destroy a swapchain until after completion of all outstanding operations on images that were acquired from the swapchain. `swapchain` and all associated `VkImage` handles are destroyed, and **must** not be acquired or used any more by the application. The memory of each `VkImage` will only be freed after that image is no longer used by the presentation engine. For example, if one image of the swapchain is being displayed in a window, the memory for that image **may** not be freed until the window is destroyed, or another swapchain is created for the window. Destroying the swapchain does not invalidate the parent `VkSurfaceKHR`, and a new swapchain **can** be created with it.

When a swapchain associated with a display surface is destroyed, if the image most recently presented to the display surface is from the swapchain being destroyed, then either any display resources modified by presenting images from any swapchain associated with the display surface **must** be reverted by the implementation to their state prior to the first present performed on one of these swapchains, or such resources **must** be left in their current state.

If `swapchain` has exclusive full-screen access, it is released before the swapchain is destroyed.
Valid Usage

- VUID-vkDestroySwapchainKHR-swapchain-01282
  All uses of presentable images acquired from swapchain must have completed execution

- VUID-vkDestroySwapchainKHR-swapchain-01283
  If VkAllocationCallbacks were provided when swapchain was created, a compatible set of callbacks must be provided here

- VUID-vkDestroySwapchainKHR-swapchain-01284
  If no VkAllocationCallbacks were provided when swapchain was created, pAllocator must be NULL

Valid Usage (Implicit)

- VUID-vkDestroySwapchainKHR-device-parameter
device must be a valid VkDevice handle

- VUID-vkDestroySwapchainKHR-swapchain-parameter
  If swapchain is not VK_NULL_HANDLE, swapchain must be a valid VkSwapchainKHR handle

- VUID-vkDestroySwapchainKHR-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroySwapchainKHR-commonparent
  Both of device, and swapchain that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkInstance

Host Synchronization

- Host access to swapchain must be externally synchronized

When the VK_KHR_display_swapchain extension is enabled, multiple swapchains that share presentable images are created by calling:

```c
// Provided by VK_KHR_display_swapchain
VkResult vkCreateSharedSwapchainsKHR(
    VkDevice device,
    uint32_t swapchainCount,
    const VkSwapchainCreateInfoKHR* pCreateInfos,
    const VkAllocationCallbacks* pAllocator,
    VkSwapchainKHR* pSwapchains);
```

- device is the device to create the swapchains for.
- swapchainCount is the number of swapchains to create.
• `pCreateInfos` is a pointer to an array of `VkSwapchainCreateInfoKHR` structures specifying the parameters of the created swapchains.

• `pAllocator` is the allocator used for host memory allocated for the swapchain objects when there is no more specific allocator available (see Memory Allocation).

• `pSwapchains` is a pointer to an array of `VkSwapchainKHR` handles in which the created swapchain objects will be returned.

`vkCreateSharedSwapchainsKHR` is similar to `vkCreateSwapchainKHR`, except that it takes an array of `VkSwapchainCreateInfoKHR` structures, and returns an array of swapchain objects.

The swapchain creation parameters that affect the properties and number of presentable images must match between all the swapchains. If the displays used by any of the swapchains do not use the same presentable image layout or are incompatible in a way that prevents sharing images, swapchain creation will fail with the result code `VK_ERROR_INCOMPATIBLE_DISPLAY_KHR`. If any error occurs, no swapchains will be created. Images presented to multiple swapchains must be re-acquired from all of them before transitioning away from `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`. After destroying one or more of the swapchains, the remaining swapchains and the presentable images can continue to be used.

Valid Usage (Implicit)

- VUID-vkCreateSharedSwapchainsKHR-device-parameter
  - `device` must be a valid `VkDevice` handle

- VUID-vkCreateSharedSwapchainsKHR-pCreateInfos-parameter
  - `pCreateInfos` must be a valid pointer to an array of `swapchainCount` valid `VkSwapchainCreateInfoKHR` structures

- VUID-vkCreateSharedSwapchainsKHR-pAllocator-parameter
  - If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

- VUID-vkCreateSharedSwapchainsKHR-pSwapchains-parameter
  - `pSwapchains` must be a valid pointer to an array of `swapchainCount` `VkSwapchainKHR` handles

- VUID-vkCreateSharedSwapchainsKHR-swapchainCount-arraylength
  - `swapchainCount` must be greater than 0

Host Synchronization

- Host access to `pCreateInfos[].surface` must be externally synchronized

- Host access to `pCreateInfos[].oldSwapchain` must be externally synchronized
Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INCOMPATIBLE_DISPLAY_KHR
- VK_ERROR_DEVICE_LOST
- VK_ERROR_SURFACE_LOST_KHR

To obtain the array of presentable images associated with a swapchain, call:

```c
// Provided by VK_KHR_swapchain
VkResult vkGetSwapchainImagesKHR(
    VkDevice device,                    // Provided by VK_KHR_swapchain
    VkSwapchainKHR swapchain,          // Provided by VK_KHR_swapchain
    uint32_t* pSwapchainImageCount,    // Provided by VK_KHR_swapchain
    VkImage* pSwapchainImages);       // Provided by VK_KHR_swapchain
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the swapchain to query.
- `pSwapchainImageCount` is a pointer to an integer related to the number of presentable images available or queried, as described below.
- `pSwapchainImages` is either `NULL` or a pointer to an array of `VkImage` handles.

If `pSwapchainImages` is `NULL`, then the number of presentable images for `swapchain` is returned in `pSwapchainImageCount`. Otherwise, `pSwapchainImageCount` must point to a variable set by the user to the number of elements in the `pSwapchainImages` array, and on return the variable is overwritten with the number of structures actually written to `pSwapchainImages`. If the value of `pSwapchainImageCount` is less than the number of presentable images for `swapchain`, at most `pSwapchainImageCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available presentable images were returned.
Valid Usage (Implicit)

- VUID-vkGetSwapchainImagesKHR-device-parameter
  *device* must be a valid *VkDevice* handle

- VUID-vkGetSwapchainImagesKHR-swapchain-parameter
  *swapchain* must be a valid *VkSwapchainKHR* handle

- VUID-vkGetSwapchainImagesKHR-pSwapchainImageCount-parameter
  *pSwapchainImageCount* must be a valid pointer to a uint32_t value

- VUID-vkGetSwapchainImagesKHR-pSwapchainImages-parameter
  If the value referenced by *pSwapchainImageCount* is not 0, and *pSwapchainImages* is not NULL,
  *pSwapchainImages* must be a valid pointer to an array of *pSwapchainImageCount* *VkImage* handles

- VUID-vkGetSwapchainImagesKHR-commonparent
  Both of *device*, and *swapchain* must have been created, allocated, or retrieved from the same *VkInstance*

Return Codes

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Note

By knowing all presentable images used in the swapchain, the application can create command buffers that reference these images prior to entering its main rendering loop.

Images returned by *vkGetSwapchainImagesKHR* are fully backed by memory before they are passed to the application. All presentable images are initially in the *VK_IMAGE_LAYOUT_UNDEFINED* layout, thus before using presentable images, the application must transition them to a valid layout for the intended use.

Further, the lifetime of presentable images is controlled by the implementation, so applications must not destroy a presentable image. See *vkDestroySwapchainKHR* for further details on the lifetime of presentable images.

Images can also be created by using *vkCreateImage* with *VkImageSwapchainCreateInfoKHR* and bound to swapchain memory using *vkBindImageMemory2KHR* with *VkBindImageMemorySwapchainInfoKHR*. These images can be used anywhere swapchain images
are used, and are useful in logical devices with multiple physical devices to create peer memory bindings of swapchain memory. These images and bindings have no effect on what memory is presented. Unlike images retrieved from `vkGetSwapchainImagesKHR`, these images must be destroyed with `vkDestroyImage`.

To acquire an available presentable image to use, and retrieve the index of that image, call:

```c
// Provided by VK_KHR_swapchain
VkResult vkAcquireNextImageKHR(
    VkDevice device,     // Provided by VK_KHR_swapchain
    VkSwapchainKHR swapchain,   // Provided by VK_KHR_swapchain
    uint64_t timeout,          // Provided by VK_KHR_swapchain
    VkSemaphore semaphore,     // Provided by VK_KHR_swapchain
    VkFence fence,            // Provided by VK_KHR_swapchain
    uint32_t* pImageIndex);   // Provided by VK_KHR_swapchain
```

- `device` is the device associated with `swapchain`.
- `swapchain` is the non-retired swapchain from which an image is being acquired.
- `timeout` specifies how long the function waits, in nanoseconds, if no image is available.
- `semaphore` is `VK_NULL_HANDLE` or a semaphore to signal.
- `fence` is `VK_NULL_HANDLE` or a fence to signal.
- `pImageIndex` is a pointer to a `uint32_t` in which the index of the next image to use (i.e. an index into the array of images returned by `vkGetSwapchainImagesKHR`) is returned.
Valid Usage

- VUID-vkAcquireNextImageKHR-swapchain-01285
  
  **swapchain** must not be in the retired state

- VUID-vkAcquireNextImageKHR-semaphore-01286
  
  If **semaphore** is not **VK_NULL_HANDLE** it must be unsignaled

- VUID-vkAcquireNextImageKHR-semaphore-01779
  
  If **semaphore** is not **VK_NULL_HANDLE** it must not have any uncompleted signal or wait operations pending

- VUID-vkAcquireNextImageKHR-fence-01287
  
  If **fence** is not **VK_NULL_HANDLE** it must be unsignaled and must not be associated with any other queue command that has not yet completed execution on that queue

- VUID-vkAcquireNextImageKHR-semaphore-01780
  
  **semaphore** and **fence** must not both be equal to **VK_NULL_HANDLE**

- VUID-vkAcquireNextImageKHR-swapchain-01802
  
  If the number of currently acquired images is greater than the difference between the number of images in **swapchain** and the value of **VkSurfaceCapabilitiesKHR::minImageCount** as returned by a call to **vkGetPhysicalDeviceSurfaceCapabilities2KHR** with the **surface** used to create **swapchain**, **timeout** must not be **UINT64_MAX**

- VUID-vkAcquireNextImageKHR-semaphore-03265
  
  **semaphore** must have a **VkSemaphoreType** of **VK_SEMAPHORE_TYPE_BINARY**
Valid Usage (Implicit)

- **VUID-vkAcquireNextImageKHR-device-parameter**
  device must be a valid VkDevice handle

- **VUID-vkAcquireNextImageKHR-swapchain-parameter**
  swapchain must be a valid VkSwapchainKHR handle

- **VUID-vkAcquireNextImageKHR-semaphore-parameter**
  If semaphore is not VK_NULL_HANDLE, semaphore must be a valid VkSemaphore handle

- **VUID-vkAcquireNextImageKHR-fence-parameter**
  If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle

- **VUID-vkAcquireNextImageKHR-pImageIndex-parameter**
  pImageIndex must be a valid pointer to a uint32_t value

- **VUID-vkAcquireNextImageKHR-semaphore-parent**
  If semaphore is a valid handle, it must have been created, allocated, or retrieved from device

- **VUID-vkAcquireNextImageKHR-fence-parent**
  If fence is a valid handle, it must have been created, allocated, or retrieved from device

- **VUID-vkAcquireNextImageKHR-commonparent**
  Both of device, and swapchain that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkInstance

Host Synchronization

- Host access to swapchain must be externally synchronized
- Host access to semaphore must be externally synchronized
- Host access to fence must be externally synchronized
Return Codes

**Success**
- VK_SUCCESS
- VK_TIMEOUT
- VK_NOT_READY
- VK_SUBOPTIMAL_KHR

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR
- VK_ERROR_SURFACE_LOST_KHR
- VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT

When successful, `vkAcquireNextImageKHR` acquires a presentable image from `swapchain` that an application can use, and sets `pImageIndex` to the index of that image within the swapchain. The presentation engine may not have finished reading from the image at the time it is acquired, so the application must use semaphore and/or fence to ensure that the image layout and contents are not modified until the presentation engine reads have completed. If semaphore is not `VK_NULL_HANDLE`, the application may assume that, once `vkAcquireNextImageKHR` returns, the semaphore signal operation referenced by `semaphore` has been submitted for execution. The order in which images are acquired is implementation-dependent, and may be different than the order the images were presented.

If `timeout` is zero, then `vkAcquireNextImageKHR` does not wait, and will either successfully acquire an image, or fail and return `VK_NOT_READY` if no image is available.

If the specified timeout period expires before an image is acquired, `vkAcquireNextImageKHR` returns `VK_TIMEOUT`. If `timeout` is `UINT64_MAX`, the timeout period is treated as infinite, and `vkAcquireNextImageKHR` will block until an image is acquired or an error occurs.

`vkAcquireNextImageKHR` should not be called if the number of images that the application has currently acquired is greater than the difference between the number of images in `swapchain` and the value of `VkSurfaceCapabilitiesKHR::minImageCount`.

If an image is acquired successfully, `vkAcquireNextImageKHR` must either return `VK_SUCCESS`, or `VK_SUBOPTIMAL_KHR` if the swapchain no longer matches the surface properties exactly, but can still be used for presentation.
If the swapchain images no longer match native surface properties, either `VK_SUBOPTIMAL_KHR` or `VK_ERROR_OUT_OF_DATE_KHR` must be returned. If `VK_ERROR_OUT_OF_DATE_KHR` is returned, no image is acquired and attempts to present previously acquired images to the swapchain will also fail with `VK_ERROR_OUT_OF_DATE_KHR`. Applications need to create a new swapchain for the surface to continue presenting if `VK_ERROR_OUT_OF_DATE_KHR` is returned.

If device loss occurs (see Lost Device) before the timeout has expired, `vkAcquireNextImageKHR` must return in finite time with either one of the allowed success codes, or `VK_ERROR_DEVICE_LOST`.

If `semaphore` is not `VK_NULL_HANDLE`, the semaphore must be unsignaled, with no signal or wait operations pending. It will become signaled when the application can use the image.

An application must wait until either the `semaphore` or `fence` is signaled before accessing the image's data.

When the presentable image will be accessed by some stage S, the recommended idiom for ensuring correct synchronization is:

- The `VkSubmitInfo` used to submit the image layout transition for execution includes `vkAcquireNextImageKHR::semaphore` in its `pWaitSemaphores` member, with the corresponding element of `pWaitDstStageMask` including S.
- The synchronization command that performs any necessary image layout transition includes S in both the `srcStageMask` and `dstStageMask`.
After a successful return, the image indicated by `pImageIndex` and its data will be unmodified compared to when it was presented.

**Note**

Exclusive ownership of presentable images corresponding to a swapchain created with `VK_SHARING_MODE_EXCLUSIVE` as defined in Resource Sharing is not altered by a call to `vkAcquireNextImageKHR`. That means upon the first acquisition from such a swapchain presentable images are not owned by any queue family, while at subsequent acquisitions the presentable images remain owned by the queue family the image was previously presented on.

The possible return values for `vkAcquireNextImageKHR` depend on the `timeout` provided:

- **VK_SUCCESS** is returned if an image became available.
- **VK_ERROR_SURFACE_LOST_KHR** is returned if the surface becomes no longer available.
- **VK_NOT_READY** is returned if `timeout` is zero and no image was available.
- **VK_TIMEOUT** is returned if `timeout` is greater than zero and less than `UINT64_MAX`, and no image became available within the time allowed.
- **VK_SUBOPTIMAL_KHR** is returned if an image became available, and the swapchain no longer matches the surface properties exactly, but can still be used to present to the surface successfully.

**Note**

This may happen, for example, if the platform surface has been resized but the platform is able to scale the presented images to the new size to produce valid surface updates. It is up to the application to decide whether it prefers to continue using the current swapchain indefinitely or temporarily in this state, or to recreate the swapchain to better match the platform surface properties.

- **VK_ERROR_OUT_OF_DATE_KHR** is returned if the surface has changed in such a way that it is no longer compatible with the swapchain, and further presentation requests using the swapchain will fail. Applications must query the new surface properties and recreate their swapchain if they wish to continue presenting to the surface.

If the native surface and presented image sizes no longer match, presentation may fail. If presentation does succeed, the mapping from the presented image to the native surface is implementation-defined. It is the application's responsibility to detect surface size changes and react appropriately. If presentation fails because of a mismatch in the surface and presented image sizes, a **VK_ERROR_OUT_OF_DATE_KHR** error will be returned.
Note

For example, consider a 4x3 window/surface that gets resized to be 3x4 (taller than wider). On some window systems, the portion of the window/surface that was previously and still is visible (the 3x3 part) will contain the same contents as before, while the remaining parts of the window will have undefined contents. Other window systems may squash/stretch the image to fill the new window size without any undefined contents, or apply some other mapping.

To acquire an available presentable image to use, and retrieve the index of that image, call:

```cpp
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
VkResult vkAcquireNextImage2KHR(
    VkDevice                                    device,
    const VkAcquireNextImageInfoKHR*            pAcquireInfo,
    uint32_t*                                   pImageIndex);
```

- `device` is the device associated with `swapchain`.
- `pAcquireInfo` is a pointer to a `VkAcquireNextImageInfoKHR` structure containing parameters of the acquire.
- `pImageIndex` is a pointer to a `uint32_t` that is set to the index of the next image to use.

**Valid Usage**

- **VUID-vkAcquireNextImage2KHR-swapchain-01803**
  If the number of currently acquired images is greater than the difference between the number of images in the `swapchain` member of `pAcquireInfo` and the value of `VkSurfaceCapabilitiesKHR::minImageCount` as returned by a call to `vkGetPhysicalDeviceSurfaceCapabilities2KHR` with the `surface` used to create `swapchain`, the `timeout` member of `pAcquireInfo` must not be `UINT64_MAX`

**Valid Usage (Implicit)**

- **VUID-vkAcquireNextImage2KHR-device-parameter**
  `device` must be a valid `VkDevice` handle
- **VUID-vkAcquireNextImage2KHR-pAcquireInfo-parameter**
  `pAcquireInfo` must be a valid pointer to a valid `VkAcquireNextImageInfoKHR` structure
- **VUID-vkAcquireNextImage2KHR-pImageIndex-parameter**
  `pImageIndex` must be a valid pointer to a `uint32_t` value
Return Codes

**Success**
- VK_SUCCESS
- VK_TIMEOUT
- VK_NOT_READY
- VK_SUBOPTIMAL_KHR

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR
- VK_ERROR_SURFACE_LOST_KHR
- VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT

The `VkAcquireNextImageInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_swapchain with VK_VERSION_1_1, VK_KHR_device_group with VK_KHR_swapchain
typedef struct VkAcquireNextImageInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkSwapchainKHR     swapchain;
    uint64_t           timeout;
    VkSemaphore        semaphore;
    VkFence            fence;
    uint32_t           deviceMask;
} VkAcquireNextImageInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchain` is a non-retired swapchain from which an image is acquired.
- `timeout` specifies how long the function waits, in nanoseconds, if no image is available.
- `semaphore` is `VK_NULL_HANDLE` or a semaphore to signal.
- `fence` is `VK_NULL_HANDLE` or a fence to signal.
- `deviceMask` is a mask of physical devices for which the swapchain image will be ready to use when the semaphore or fence is signaled.

If `vkAcquireNextImageInfoKHR` is used, the device mask is considered to include all physical devices in the logical device.
Note

vkAcquireNextImage2KHR signals at most one semaphore, even if the application requests waiting for multiple physical devices to be ready via the deviceMask. However, only a single physical device can wait on that semaphore, since the semaphore becomes unsignaled when the wait succeeds. For other physical devices to wait for the image to be ready, it is necessary for the application to submit semaphore signal operation(s) to that first physical device to signal additional semaphore(s) after the wait succeeds, which the other physical device(s) can wait upon.

Valid Usage

- VUID-VkAcquireNextImageInfoKHR-swapchain-01675
  swapchain must not be in the retired state

- VUID-VkAcquireNextImageInfoKHR-semaphore-01288
  If semaphore is not VK_NULL_HANDLE it must be unsignaled

- VUID-VkAcquireNextImageInfoKHR-semaphore-01781
  If semaphore is not VK_NULL_HANDLE it must not have any uncompleted signal or wait operations pending

- VUID-VkAcquireNextImageInfoKHR-fence-01289
  If fence is not VK_NULL_HANDLE it must be unsignaled and must not be associated with any other queue command that has not yet completed execution on that queue

- VUID-VkAcquireNextImageInfoKHR-semaphore-01782
  semaphore and fence must not both be equal to VK_NULL_HANDLE

- VUID-VkAcquireNextImageInfoKHR-deviceMask-01290
  deviceMask must be a valid device mask

- VUID-VkAcquireNextImageInfoKHR-deviceMask-01291
  deviceMask must not be zero

- VUID-VkAcquireNextImageInfoKHR-semaphore-03266
  semaphore must have a VkSemaphoreType of VK_SEMAPHORE_TYPE_BINARY
Valid Usage (Implicit)

- VUID-VkAcquireNextImageInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR

- VUID-VkAcquireNextImageInfoKHR-pNext-pNext
  pNext must be NULL

- VUID-VkAcquireNextImageInfoKHR-swapchain-parameter
  swapchain must be a valid VkSwapchainKHR handle

- VUID-VkAcquireNextImageInfoKHR-semaphore-parameter
  If semaphore is not VK_NULL_HANDLE, semaphore must be a valid VkSemaphore handle

- VUID-VkAcquireNextImageInfoKHR-fence-parameter
  If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle

- VUID-VkAcquireNextImageInfoKHR-commonparent
  Each of fence, semaphore, and swapchain that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkInstance

Host Synchronization

- Host access to swapchain must be externally synchronized
- Host access to semaphore must be externally synchronized
- Host access to fence must be externally synchronized

After queueing all rendering commands and transitioning the image to the correct layout, to queue an image for presentation, call:

```c
// Provided by VK_KHR_swapchain
VkResult vkQueuePresentKHR(
    VkQueue                                     queue,
    const VkPresentInfoKHR*                     pPresentInfo);
```

- queue is a queue that is capable of presentation to the target surface's platform on the same device as the image's swapchain.

- pPresentInfo is a pointer to a VkPresentInfoKHR structure specifying parameters of the presentation.

Note

There is no requirement for an application to present images in the same order that they were acquired - applications can arbitrarily present any image that is currently acquired.
Valid Usage

- **VUID-vkQueuePresentKHR-pSwapchains-01292**
  Each element of `pSwapchains` member of `pPresentInfo` must be a swapchain that is created for a surface for which presentation is supported from `queue` as determined using a call to `vkGetPhysicalDeviceSurfaceSupportKHR`.

- **VUID-vkQueuePresentKHR-pSwapchains-01293**
  If more than one member of `pSwapchains` was created from a display surface, all display surfaces referenced that refer to the same display must use the same display mode.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-01294**
  When a semaphore wait operation referring to a binary semaphore defined by the elements of the `pWaitSemaphores` member of `pPresentInfo` executes on `queue`, there must be no other queues waiting on the same semaphore.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-01295**
  All elements of the `pWaitSemaphores` member of `pPresentInfo` must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-03267**
  All elements of the `pWaitSemaphores` member of `pPresentInfo` must be created with a `VkSemaphoreType` of `VK_SEMAPHORE_TYPE_BINARY`.

- **VUID-vkQueuePresentKHR-pWaitSemaphores-03268**
  All elements of the `pWaitSemaphores` member of `pPresentInfo` must reference a semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) must have also been submitted for execution.

Any writes to memory backing the images referenced by the `pImageIndices` and `pSwapchains` members of `pPresentInfo`, that are available before `vkQueuePresentKHR` is executed, are automatically made visible to the read access performed by the presentation engine. This automatic visibility operation for an image happens-after the semaphore signal operation, and happens-before the presentation engine accesses the image.

Queueing an image for presentation defines a set of *queue operations*, including waiting on the semaphores and submitting a presentation request to the presentation engine. However, the scope of this set of queue operations does not include the actual processing of the image by the presentation engine.

**Note**

The origin of the native orientation of the surface coordinate system is not specified in the Vulkan specification; it depends on the platform. For most platforms the origin is by default upper-left, meaning the pixel of the presented `VkImage` at coordinates (0,0) would appear at the upper left pixel of the platform surface (assuming `VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR`, and the display standing the right way up).

If `vkQueuePresentKHR` fails to enqueue the corresponding set of queue operations, it may return `VK_ERROR_OUT_OF_HOST_MEMORY` or `VK_ERROR_OUT_OF_DEVICE_MEMORY`. If it does, the implementation must...
ensure that the state and contents of any resources or synchronization primitives referenced is unaffected by the call or its failure.

If `vkQueuePresentKHR` fails in such a way that the implementation is unable to make that guarantee, the implementation **must** return `VK_ERROR_DEVICE_LOST`.

However, if the presentation request is rejected by the presentation engine with an error `VK_ERROR_OUT_OF_DATE_KHR`, `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT`, or `VK_ERROR_SURFACE_LOST_KHR`, the set of queue operations are still considered to be enqueued and thus any semaphore wait operation specified in `VkPresentInfoKHR` will execute when the corresponding queue operation is complete.

If any swapchain member of `pPresentInfo` was created with `VK_FULL_SCREEN_EXCLUSIVE_APPLICATION_CONTROLLED_EXT`, `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT` will be returned if that swapchain does not have exclusive full-screen access, possibly for implementation-specific reasons outside of the application's control.

### Valid Usage (Implicit)

- **VUID-vkQueuePresentKHR-queue-parameter**
  
  queue **must** be a valid `VkQueue` handle

- **VUID-vkQueuePresentKHR-pPresentInfo-parameter**
  
  `pPresentInfo` **must** be a valid pointer to a valid `VkPresentInfoKHR` structure

### Host Synchronization

- Host access to queue **must** be externally synchronized
- Host access to `pPresentInfo->pWaitSemaphores[]` **must** be externally synchronized
- Host access to `pPresentInfo->pSwapchains[]` **must** be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>
Return Codes

Success
- VK_SUCCESS
- VK_SUBOPTIMAL_KHR

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_DEVICE_LOST
- VK_ERROR_OUT_OF_DATE_KHR
- VK_ERROR_SURFACE_LOST_KHR
- VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT

The `VkPresentInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_swapchain
typedef struct VkPresentInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    uint32_t swapchainCount;
    const VkSwapchainKHR* pSwapchains;
    const uint32_t* pImageIndices;
    VkResult* pResults;
} VkPresentInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `waitSemaphoreCount` is the number of semaphores to wait for before issuing the present request. The number may be zero.
- `pWaitSemaphores` is NULL or a pointer to an array of `VkSemaphore` objects with `waitSemaphoreCount` entries, and specifies the semaphores to wait for before issuing the present request.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pSwapchains` is a pointer to an array of `VkSwapchainKHR` objects with `swapchainCount` entries. A given swapchain must not appear in this list more than once.
- `pImageIndices` is a pointer to an array of indices into the array of each swapchain's presentable images, with `swapchainCount` entries. Each entry in this array identifies the image to present on the corresponding entry in the `pSwapchains` array.
- `pResults` is a pointer to an array of `VkResult` typed elements with `swapchainCount` entries.
Applications that do not need per-swapchain results can use `NULL` for `pResults`. If non-`NULL`, each entry in `pResults` will be set to the `VkResult` for presenting the swapchain corresponding to the same index in `pSwapchains`.

Before an application can present an image, the image's layout must be transitioned to the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout, or for a shared presentable image the `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout.

**Note**

When transitioning the image to `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` or `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`, there is no need to delay subsequent processing, or perform any visibility operations (as `vkQueuePresentKHR` performs automatic visibility operations). To achieve this, the `dstAccessMask` member of the `VkImageMemoryBarrier` should be set to `0`, and the `dstStageMask` parameter should be set to `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`.

**Valid Usage**

- **VUID-VkPresentInfoKHR-pImageIndices-01430**

  Each element of `pImageIndices` must be the index of a presentable image acquired from the swapchain specified by the corresponding element of the `pSwapchains` array, and the presented image subresource must be in the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` or `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR` layout at the time the operation is executed on a `VkDevice`. 
Valid Usage (Implicit)

- **VUID-VkPresentInfoKHR-sType-sType**
  
  sType must be VK_STRUCTURE_TYPE_PRESENT_INFO_KHR

- **VUID-VkPresentInfoKHR-pNext-pNext**
  
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkDeviceGroupPresentInfoKHR, VkDisplayPresentInfoKHR, VkPresentFrameTokenGGP, VkPresentRegionsKHR, or VkPresentTimesInfoGOOGLE

- **VUID-VkPresentInfoKHR-sType-unique**
  
  The sType value of each struct in the pNext chain must be unique

- **VUID-VkPresentInfoKHR-pWaitSemaphores-parameter**
  
  If waitSemaphoreCount is not 0, pWaitSemaphores must be a valid pointer to an array of waitSemaphoreCount valid VkSemaphore handles

- **VUID-VkPresentInfoKHR-pSwapchains-parameter**
  
  pSwapchains must be a valid pointer to an array of swapchainCount valid VkSwapchainKHR handles

- **VUID-VkPresentInfoKHR-pImageIndices-parameter**
  
  pImageIndices must be a valid pointer to an array of swapchainCount uint32_t values

- **VUID-VkPresentInfoKHR-pResults-parameter**
  
  If pResults is not NULL, pResults must be a valid pointer to an array of swapchainCount VkResult values

- **VUID-VkPresentInfoKHR-swapchainCount-arraylength**
  
  swapchainCount must be greater than 0

- **VUID-VkPresentInfoKHR-commonparent**
  
  Both of the elements of pSwapchains, and the elements of pWaitSemaphores that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same VkInstance

When the VK_KHR_incremental_present extension is enabled, additional fields can be specified that allow an application to specify that only certain rectangular regions of the presentable images of a swapchain are changed. This is an optimization hint that a presentation engine may use to only update the region of a surface that is actually changing. The application still must ensure that all pixels of a presented image contain the desired values, in case the presentation engine ignores this hint. An application can provide this hint by adding a VkPresentRegionsKHR structure to the pNext chain of the VkPresentInfoKHR structure.

The VkPresentRegionsKHR structure is defined as:
typedef struct VkPresentRegionsKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t swapchainCount;
    const VkPresentRegionKHR* pRegions;
} VkPresentRegionsKHR;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pRegions` is `NULL` or a pointer to an array of `VkPresentRegionKHR` elements with `swapchainCount` entries. If not `NULL`, each element of `pRegions` contains the region that has changed since the last present to the swapchain in the corresponding entry in the `VkPresentInfoKHR::pSwapchains` array.

**Valid Usage**

- **VUID-VkPresentRegionsKHR-swapchainCount-01260**
  
  `swapchainCount` must be the same value as `VkPresentInfoKHR::swapchainCount`, where `VkPresentInfoKHR` is included in the `pNext` chain of this `VkPresentRegionsKHR` structure

**Valid Usage (Implicit)**

- **VUID-VkPresentRegionsKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR`

- **VUID-VkPresentRegionsKHR-pRegions-parameter**
  
  If `pRegions` is not `NULL`, `pRegions` must be a valid pointer to an array of `swapchainCount` valid `VkPresentRegionKHR` structures

- **VUID-VkPresentRegionsKHR-swapchainCount-arraylength**
  
  `swapchainCount` must be greater than 0

For a given image and swapchain, the region to present is specified by the `VkPresentRegionKHR` structure, which is defined as:

```c
// Provided by VK_KHR_incremental_present
typedef struct VkPresentRegionKHR {
    uint32_t rectangleCount;
    const VkRectLayerKHR* pRectangles;
} VkPresentRegionKHR;
```

- `rectangleCount` is the number of rectangles in `pRectangles`, or zero if the entire image has changed and should be presented.
- `pRectangles` is either `NULL` or a pointer to an array of `VkRectLayerKHR` structures. The `VkRectLayerKHR` structure is the framebuffer coordinates, plus layer, of a portion of a presentable image that has changed and must be presented. If non-`NULL`, each entry in `pRectangles` is a rectangle of the given image that has changed since the last image was presented to the given swapchain. The rectangles must be specified relative to `VkSurfaceCapabilitiesKHR::currentTransform`, regardless of the swapchain’s `preTransform`. The presentation engine will apply the `preTransform` transformation to the rectangles, along with any further transformation it applies to the image content.

**Valid Usage (Implicit)**

- `VUID-VkPresentRegionKHR-pRectangles-parameter` If `rectangleCount` is not 0, and `pRectangles` is not `NULL`, `pRectangles` must be a valid pointer to an array of `rectangleCount` valid `VkRectLayerKHR` structures.

The `VkRectLayerKHR` structure is defined as:

```
// Provided by VK_KHR_incremental_present
typedef struct VkRectLayerKHR {
    VkOffset2D     offset;
    VkExtent2D     extent;
    uint32_t       layer;
} VkRectLayerKHR;
```

- `offset` is the origin of the rectangle, in pixels.
- `extent` is the size of the rectangle, in pixels.
- `layer` is the layer of the image. For images with only one layer, the value of `layer` must be 0.

**Valid Usage**

- `VUID-VkRectLayerKHR-offset-04864` The sum of `offset` and `extent`, after being transformed according to the `preTransform` member of the `VkSwapchainCreateInfoKHR` structure, must be no greater than the `imageExtent` member of the `VkSwapchainCreateInfoKHR` structure passed to `vkCreateSwapchainKHR`.
- `VUID-VkRectLayerKHR-layer-01262` `layer` must be less than the `imageArrayLayers` member of the `VkSwapchainCreateInfoKHR` structure passed to `vkCreateSwapchainKHR`.

Some platforms allow the size of a surface to change, and then scale the pixels of the image to fit the surface. `VkRectLayerKHR` specifies pixels of the swapchain’s image(s), which will be constant for the life of the swapchain.

When the `VK_KHR_display_swapchain` extension is enabled additional fields can be specified when
presenting an image to a swapchain by setting `VkPresentInfoKHR::pNext` to point to a `VkDisplayPresentInfoKHR` structure.

The `VkDisplayPresentInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_display_swapchain
typedef struct VkDisplayPresentInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkRect2D srcRect;
    VkRect2D dstRect;
    VkBool32 persistent;
} VkDisplayPresentInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **srcRect** is a rectangular region of pixels to present. It **must** be a subset of the image being presented. If `VkDisplayPresentInfoKHR` is not specified, this region will be assumed to be the entire presentable image.
- **dstRect** is a rectangular region within the visible region of the swapchain’s display mode. If `VkDisplayPresentInfoKHR` is not specified, this region will be assumed to be the entire visible region of the swapchain’s mode. If the specified rectangle is a subset of the display mode’s visible region, content from display planes below the swapchain’s plane will be visible outside the rectangle. If there are no planes below the swapchain’s, the area outside the specified rectangle will be black. If portions of the specified rectangle are outside of the display’s visible region, pixels mapping only to those portions of the rectangle will be discarded.
- **persistent**: If this is `VK_TRUE`, the display engine will enable buffered mode on displays that support it. This allows the display engine to stop sending content to the display until a new image is presented. The display will instead maintain a copy of the last presented image. This allows less power to be used, but **may** increase presentation latency. If `VkDisplayPresentInfoKHR` is not specified, persistent mode will not be used.

If the extent of the **srcRect** and **dstRect** are not equal, the presented pixels will be scaled accordingly.
Valid Usage

- **VUID-VkDisplayPresentInfoKHR-srcRect-01257**
  
  `srcRect` must specify a rectangular region that is a subset of the image being presented.

- **VUID-VkDisplayPresentInfoKHR-dstRect-01258**
  
  `dstRect` must specify a rectangular region that is a subset of the `visibleRegion` parameter of the display mode the swapchain being presented uses.

- **VUID-VkDisplayPresentInfoKHR-persistentContent-01259**
  
  If the `persistentContent` member of the `VkDisplayPropertiesKHR` structure returned by `vkGetPhysicalDeviceDisplayPropertiesKHR` for the display the present operation targets is `VK_FALSE`, then `persistent` must be `VK_FALSE`.

Valid Usage (Implicit)

- **VUID-VkDisplayPresentInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR`.

If the `pNext` chain of `VkPresentInfoKHR` includes a `VkDeviceGroupPresentInfoKHR` structure, then that structure includes an array of device masks and a device group present mode.

The `VkDeviceGroupPresentInfoKHR` structure is defined as:

```c
typedef struct VkDeviceGroupPresentInfoKHR {
    VkStructureType                        sType;      
    const void*                            pNext;      
    uint32_t                               swapchainCount;      
    const uint32_t*                        pDeviceMasks;  
    VkDeviceGroupPresentModeFlagBitsKHR    mode;      
} VkDeviceGroupPresentInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchainCount` is zero or the number of elements in `pDeviceMasks`.
- `pDeviceMasks` is a pointer to an array of device masks, one for each element of `VkPresentInfoKHR::pSwapchains`.
- `mode` is the device group present mode that will be used for this present.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`, then each element of `pDeviceMasks` selects which instance of the swapchain image is presented. Each element of `pDeviceMasks` must have exactly one bit set, and the corresponding physical device must have a presentation engine as reported by `VkDeviceGroupPresentCapabilitiesKHR`.

1860 | Chapter 33. Window System Integration (WSI)
If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR`, then each element of `pDeviceMasks` selects which instance of the swapchain image is presented. Each element of `pDeviceMasks` must have exactly one bit set, and some physical device in the logical device must include that bit in its `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR`, then each element of `pDeviceMasks` selects which instances of the swapchain image are component-wise summed and the sum of those images is presented. If the sum in any component is outside the representable range, the value of that component is undefined. Each element of `pDeviceMasks` must have a value for which all set bits are set in one of the elements of `VkDeviceGroupPresentCapabilitiesKHR::presentMask`.

If `mode` is `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR`, then each element of `pDeviceMasks` selects which instance(s) of the swapchain images are presented. For each bit set in each element of `pDeviceMasks`, the corresponding physical device must have a presentation engine as reported by `VkDeviceGroupPresentCapabilitiesKHR`.

If `VkDeviceGroupPresentInfoKHR` is not provided or `swapchainCount` is zero then the masks are considered to be 1. If `VkDeviceGroupPresentInfoKHR` is not provided, `mode` is considered to be `VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR`. 
Valid Usage

- VUID-VkDeviceGroupPresentInfoKHR-swapchainCount-01297
  swapchainCount must equal 0 or VkPresentInfoKHR::swapchainCount

- VUID-VkDeviceGroupPresentInfoKHR-mode-01298
  If mode is VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_BIT_KHR, then each element of pDeviceMasks
  must have exactly one bit set, and the corresponding element of
  VkDeviceGroupPresentCapabilitiesKHR::presentMask must be non-zero

- VUID-VkDeviceGroupPresentInfoKHR-mode-01299
  If mode is VK_DEVICE_GROUP_PRESENT_MODE_REMOTE_BIT_KHR, then each element of pDeviceMasks
  must have exactly one bit set, and some physical device in the logical device must include
  that bit in its VkDeviceGroupPresentCapabilitiesKHR::presentMask

- VUID-VkDeviceGroupPresentInfoKHR-mode-01300
  If mode is VK_DEVICE_GROUP_PRESENT_MODE_SUM_BIT_KHR, then each element of pDeviceMasks
  must have a value for which all set bits are set in one of the elements of
  VkDeviceGroupPresentCapabilitiesKHR::presentMask

- VUID-VkDeviceGroupPresentInfoKHR-mode-01301
  If mode is VK_DEVICE_GROUP_PRESENT_MODE_LOCAL_MULTI_DEVICE_BIT_KHR, then for each bit set
  in each element of pDeviceMasks, the corresponding element of
  VkDeviceGroupPresentCapabilitiesKHR::presentMask must be non-zero

- VUID-VkDeviceGroupPresentInfoKHR-pDeviceMasks-01302
  The value of each element of pDeviceMasks must be equal to the device mask passed in
  VkAcquireNextImageInfoKHR::deviceMask when the image index was last acquired

- VUID-VkDeviceGroupPresentInfoKHR-mode-01303
  mode must have exactly one bit set, and that bit must have been included in
  VkDeviceGroupSwapchainCreateInfoKHR::modes

Valid Usage (Implicit)

- VUID-VkDeviceGroupPresentInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR

- VUID-VkDeviceGroupPresentInfoKHR-pDeviceMasks-parameter
  If swapchainCount is not 0, pDeviceMasks must be a valid pointer to an array of
  swapchainCount uint32_t values

- VUID-VkDeviceGroupPresentInfoKHR-mode-parameter
  mode must be a valid VkDeviceGroupPresentModeFlagBitsKHR value

When the VK_GOOGLE_display_timing extension is enabled, additional fields can be specified that
allow an application to specify the earliest time that an image should be displayed. This allows an
application to avoid stutter that is caused by an image being displayed earlier than planned. Such
stuttering can occur with both fixed and variable-refresh-rate displays, because stuttering occurs
when the geometry is not correctly positioned for when the image is displayed. An application can
instruct the presentation engine that an image should not be displayed earlier than a specified time by adding a `VkPresentTimesInfoGOOGLE` structure to the `pNext` chain of the `VkPresentInfoKHR` structure.

The `VkPresentTimesInfoGOOGLE` structure is defined as:

```c
// Provided by VK_GOOGLE_display_timing
typedef struct VkPresentTimesInfoGOOGLE {
    VkStructureType        sType;
    const void*            pNext;
    uint32_t                swapchainCount;
    const VkPresentTimeGOOGLE*    pTimes;
} VkPresentTimesInfoGOOGLE;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pTimes` is `NULL` or a pointer to an array of `VkPresentTimeGOOGLE` elements with `swapchainCount` entries. If not `NULL`, each element of `pTimes` contains the earliest time to present the image corresponding to the entry in the `VkPresentInfoKHR::pImageIndices` array.

### Valid Usage

- `VUID-VkPresentTimesInfoGOOGLE-swapchainCount-01247`
  
  `swapchainCount` must be the same value as `VkPresentInfoKHR::swapchainCount`, where `VkPresentInfoKHR` is included in the `pNext` chain of this `VkPresentTimesInfoGOOGLE` structure.

### Valid Usage (Implicit)

- `VUID-VkPresentTimesInfoGOOGLE-sType-sType`
  
  `sType` must be `VK_STRUCTURE_TYPE_PRESENT_TIMES_INFO_GOOGLE`.

- `VUID-VkPresentTimesInfoGOOGLE-pTimes-parameter`
  
  If `pTimes` is not `NULL`, `pTimes` must be a valid pointer to an array of `VkPresentTimeGOOGLE` structures.

- `VUID-VkPresentTimesInfoGOOGLE-swapchainCount-arraylength`
  
  `swapchainCount` must be greater than 0.

The `VkPresentTimeGOOGLE` structure is defined as:

```c
// Provided by VK_GOOGLE_display_timing
typedef struct VkPresentTimeGOOGLE {
    uint32_t    presentID;
    uint64_t    desiredPresentTime;
} VkPresentTimeGOOGLE;
```
• presentID is an application-provided identification value, that can be used with the results of vkGetPastPresentationTimingGOOGLE, in order to uniquely identify this present. In order to be useful to the application, it should be unique within some period of time that is meaningful to the application.

• desiredPresentTime specifies that the image given should not be displayed to the user any earlier than this time. desiredPresentTime is a time in nanoseconds, relative to a monotonically-increasing clock (e.g. CLOCK_MONOTONIC (see clock_gettime(2)) on Android and Linux). A value of zero specifies that the presentation engine may display the image at any time. This is useful when the application desires to provide presentID, but does not need a specific desiredPresentTime.

When the VK_GGP_frame_token extension is enabled, a Google Games Platform frame token can be specified when presenting an image to a swapchain by adding a VkPresentFrameTokenGGP structure to the pNext chain of the VkPresentInfoKHR structure.

The VkPresentFrameTokenGGP structure is defined as:

```c
// Provided by VK_GGP_frame_token
typedef struct VkPresentFrameTokenGGP {
    VkStructureType    sType;
    const void*        pNext;
    GgpFrameToken      frameToken;
} VkPresentFrameTokenGGP;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• frameToken is the Google Games Platform frame token.

**Valid Usage**

- VUID-VkPresentFrameTokenGGP-frameToken-02680
  frameToken must be a valid GgpFrameToken

**Valid Usage (Implicit)**

- VUID-VkPresentFrameTokenGGP-sType-sType
  sType must be VK_STRUCTURE_TYPE_PRESENT_FRAME_TOKEN_GGP

vkQueuePresentKHR, releases the acquisition of the images referenced by imageIndices. The queue family corresponding to the queue vkQueuePresentKHR is executed on must have ownership of the presented images as defined in Resource Sharing. vkQueuePresentKHR does not alter the queue family ownership, but the presented images must not be used again before they have been reacquired using vkAcquireNextImageKHR.

The processing of the presentation happens in issue order with other queue operations, but
semaphores have to be used to ensure that prior rendering and other commands in the specified queue complete before the presentation begins. The presentation command itself does not delay processing of subsequent commands on the queue, however, presentation requests sent to a particular queue are always performed in order. Exact presentation timing is controlled by the semantics of the presentation engine and native platform in use.

If an image is presented to a swapchain created from a display surface, the mode of the associated display will be updated, if necessary, to match the mode specified when creating the display surface. The mode switch and presentation of the specified image will be performed as one atomic operation.

The result codes `VK_ERROR_OUT_OF_DATE_KHR` and `VK_SUBOPTIMAL_KHR` have the same meaning when returned by `vkQueuePresentKHR` as they do when returned by `vkAcquireNextImageKHR`. If multiple swapchains are presented, the result code is determined applying the following rules in order:

- If the device is lost, `VK_ERROR_DEVICE_LOST` is returned.
- If any of the target surfaces are no longer available the error `VK_ERROR_SURFACE_LOST_KHR` is returned.
- If any of the presents would have a result of `VK_ERROR_OUT_OF_DATE_KHR` if issued separately then `VK_ERROR_OUT_OF_DATE_KHR` is returned.
- If any of the presents would have a result of `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT` if issued separately then `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT` is returned.
- If any of the presents would have a result of `VK_SUBOPTIMAL_KHR` if issued separately then `VK_SUBOPTIMAL_KHR` is returned.
- Otherwise `VK_SUCCESS` is returned.

Presentation is a read-only operation that will not affect the content of the presentable images. Upon reacquiring the image and transitioning it away from the `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` layout, the contents will be the same as they were prior to transitioning the image to the present source layout and presenting it. However, if a mechanism other than Vulkan is used to modify the platform window associated with the swapchain, the content of all presentable images in the swapchain becomes undefined.

Note
The application can continue to present any acquired images from a retired swapchain as long as the swapchain has not entered a state that causes `vkQueuePresentKHR` to return `VK_ERROR_OUT_OF_DATE_KHR`.

33.10. Hdr Metadata

To improve color reproduction of content it is useful to have information that can be used to better reproduce the colors as seen on the reference monitor. That information can be provided to an implementation by calling `vkSetHdrMetadataEXT`. The metadata will be applied to the specified `VkSwapchainKHR` objects at the next `vkQueuePresentKHR` call using that `VkSwapchainKHR` object. The metadata will persist until a subsequent `vkSetHdrMetadataEXT` changes it. The definitions below are from the associated SMPTE 2086, CTA 861.3 and CIE 15:2004 specifications.
The definition of `vkSetHdrMetadataEXT` is:

```c
// Provided by VK_EXT_hdr_metadata
void vkSetHdrMetadataEXT(
    VkDevice                                    device,               
    uint32_t                                    swapchainCount,      
    const VkSwapchainKHR*                       pSwapchains,         
    const VkHdrMetadataEXT*                     pMetadata);          
```

- `device` is the logical device where the swapchain(s) were created.
- `swapchainCount` is the number of swapchains included in `pSwapchains`.
- `pSwapchains` is a pointer to an array of `swapchainCount` `VkSwapchainKHR` handles.
- `pMetadata` is a pointer to an array of `swapchainCount` `VkHdrMetadataEXT` structures.

**Valid Usage (Implicit)**

- VUID-vkSetHdrMetadataEXT-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkSetHdrMetadataEXT-pSwapchains-parameter
  - `pSwapchains` must be a valid pointer to an array of `swapchainCount` valid `VkSwapchainKHR` handles
- VUID-vkSetHdrMetadataEXT-pMetadata-parameter
  - `pMetadata` must be a valid pointer to an array of `swapchainCount` valid `VkHdrMetadataEXT` structures
- VUID-vkSetHdrMetadataEXT-swapchainCount-arraylength
  - `swapchainCount` must be greater than 0
- VUID-vkSetHdrMetadataEXT-commonparent
  - Both of `device`, and the elements of `pSwapchains` must have been created, allocated, or retrieved from the same `VkInstance`

```
// Provided by VK_EXT_hdr_metadata
typedef struct VkXYColorEXT {
    float x;
    float y;
} VkXYColorEXT;
```

Chromaticity coordinates x and y are as specified in CIE 15:2004 “Calculation of chromaticity coordinates” (Section 7.3) and are limited to between 0 and 1 for real colors for the reference monitor.
// Provided by VK_EXT_hdr_metadata

typedef struct VkHdrMetadataEXT {
    VkStructureType    sType;
    const void*        pNext;
    VkXYColorEXT       displayPrimaryRed;
    VkXYColorEXT       displayPrimaryGreen;
    VkXYColorEXT       displayPrimaryBlue;
    VkXYColorEXT       whitePoint;
    float              maxLuminance;
    float              minLuminance;
    float              maxContentLightLevel;
    float              maxFrameAverageLightLevel;
} VkHdrMetadataEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **displayPrimaryRed** is the reference monitor's red primary in chromaticity coordinates
- **displayPrimaryGreen** is the reference monitor's green primary in chromaticity coordinates
- **displayPrimaryBlue** is the reference monitor's blue primary in chromaticity coordinates
- **whitePoint** is the reference monitor's white-point in chromaticity coordinates
- **maxLuminance** is the maximum luminance of the reference monitor in nits
- **minLuminance** is the minimum luminance of the reference monitor in nits
- **maxContentLightLevel** is content's maximum luminance in nits
- **maxFrameAverageLightLevel** is the maximum frame average light level in nits

### Valid Usage (Implicit)

- **VUID-VkHdrMetadataEXT-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_HDR_METADATA_EXT**
  - **VUID-VkHdrMetadataEXT-pNext-pNext**
  - **pNext** must be NULL

---

**Note**

The validity and use of this data is outside the scope of Vulkan.
Chapter 34. Deferred Host Operations

Certain Vulkan commands are inherently expensive for the host CPU to execute. It is often desirable to offload such work onto background threads, and to parallelize the work across multiple CPUs. The concept of deferred operations allows applications and drivers to coordinate the execution of expensive host commands using an application-managed thread pool.

The VK_KHR_deferred_host_operations extension defines the infrastructure and usage patterns for deferrable commands, but does not specify any commands as deferrable. This is left to additional dependant extensions. Commands must not be deferred unless the deferral is specifically allowed by another extension which depends on VK_KHR_deferred_host_operations. This specification will refer to such extensions as deferral extensions.

34.1. Requesting Deferral

When an application requests an operation deferral, the implementation may defer the operation. When deferral is requested and the implementation defers any operation, the implementation must return VK_OPERATION_DEFERRED_KHR as the success code if no errors occurred. When deferral is requested, the implementation should defer the operation when the workload is significant, however if the implementation chooses not to defer any of the requested operations and instead executes all of them immediately, the implementation must return VK_OPERATION_NOT_DEFERRED_KHR as the success code if no errors occurred.

A deferred operation is created complete with an initial result value of VK_SUCCESS. The deferred operation becomes pending when an operation has been successfully deferred with that deferred operation object.

A deferred operation is considered pending until the deferred operation completes. A pending deferred operation becomes complete when it has been fully executed by one or more threads. Pending deferred operations will never complete until they are joined by an application thread, using vkDeferredOperationJoinKHR. Applications can join multiple threads to the same deferred operation, enabling concurrent execution of subtasks within that operation.

The application can query the status of a VkDeferredOperationKHR using the vkGetDeferredOperationMaxConcurrencyKHR or vkGetDeferredOperationResultKHR commands.

From the perspective of other commands, parameters to the original command that are externally synchronized must not be accessed before the deferred operation completes, and the result of the deferred operation (e.g. object creation) is not considered complete until the deferred operation completes.

If the deferred operation is one which creates an object or otherwise returns a value to the application, output parameters may be accessed by the implementation up until the deferred operation is complete. The application must not access the return parameters until the deferred operation is complete. When the deferred operation is complete, the application should call vkGetDeferredOperationResultKHR to obtain the result of the operation. Results are returned in the same manner as the original command would have returned them, as it would when not deferred and returning the result obtained from vkGetDeferredOperationResultKHR.
34.2. Deferred Host Operations API

The `VkDeferredOperationKHR` handle is defined as:

```c
// Provided by VK_KHR_deferred_host_operations
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDeferredOperationKHR)
```

This handle refers to a tracking structure which manages the execution state for a deferred command.

To construct the tracking object for a deferred command, call:

```c
// Provided by VK_KHR_deferred_host_operations
VkResult vkCreateDeferredOperationKHR(
    VkDevice                                    device,                        
    const VkAllocationCallbacks*                pAllocator,                   
    VkDeferredOperationKHR*                     pDeferredOperation);
```

- `device` is the device which owns `operation`.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pDeferredOperation` is a pointer to a handle in which the created `VkDeferredOperationKHR` is returned.

### Valid Usage (Implicit)

- **VUID-vkCreateDeferredOperationKHR-device-parameter**
  - `device` must be a valid `VkDevice` handle
- **VUID-vkCreateDeferredOperationKHR-pAllocator-parameter**
  - If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **VUID-vkCreateDeferredOperationKHR-pDeferredOperation-parameter**
  - `pDeferredOperation` must be a valid pointer to a `VkDeferredOperationKHR` handle

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`

To assign a thread to a deferred operation, call:
```c
// Provided by VK_KHR_deferred_host_operations
VkResult vkDeferredOperationJoinKHR(
    VkDevice device,
    VkDeferredOperationKHR operation);
```

- **device** is the device which owns **operation**.
- **operation** is the deferred operation that the calling thread should work on.

The `vkDeferredOperationJoinKHR` command will execute a portion of the deferred operation on the calling thread.

The return value will be one of the following:

- A return value of `VK_SUCCESS` indicates that **operation** is complete. The application **should** use `vkGetDeferredOperationResultKHR` to retrieve the result of **operation**.
- A return value of `VK_THREAD_DONE_KHR` indicates that the deferred operation is not complete, but there is no work remaining to assign to threads. Future calls to `vkDeferredOperationJoinKHR` are not necessary and will simply harm performance. This situation **may** occur when other threads executing `vkDeferredOperationJoinKHR` are about to complete **operation**, and the implementation is unable to partition the workload any further.
- A return value of `VK_THREAD_IDLE_KHR` indicates that the deferred operation is not complete, and there is no work for the thread to do at the time of the call. This situation **may** occur if the operation encounters a temporary reduction in parallelism. By returning `VK_THREAD_IDE_KHR`, the implementation is signaling that it expects that more opportunities for parallelism will emerge as execution progresses, and that future calls to `vkDeferredOperationJoinKHR` can be beneficial. In the meantime, the application **can** perform other work on the calling thread.

Implementations **must** guarantee forward progress by enforcing the following invariants:

1. If only one thread has invoked `vkDeferredOperationJoinKHR` on a given operation, that thread **must** execute the operation to completion and return `VK_SUCCESS`.
2. If multiple threads have concurrently invoked `vkDeferredOperationJoinKHR` on the same operation, then at least one of them **must** complete the operation and return `VK_SUCCESS`.

### Valid Usage (Implicit)

- **VUID-vkDeferredOperationJoinKHR-device-parameter**
  
  **device** **must** be a valid **VkDevice** handle

- **VUID-vkDeferredOperationJoinKHR-operation-parameter**
  
  **operation** **must** be a valid **VkDeferredOperationKHR** handle

- **VUID-vkDeferredOperationJoinKHR-operation-parent**
  
  **operation** **must** have been created, allocated, or retrieved from **device**
Return Codes

**Success**
- VK_SUCCESS
- VK_THREAD_DONE_KHR
- VK_THREAD_IDLE_KHR

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

When a deferred operation is completed, the application can destroy the tracking object by calling:

```c
// Provided by VK_KHR_deferred_host_operations
void vkDestroyDeferredOperationKHR(
    VkDevice device,
    VkDeferredOperationKHR operation,
    const VkAllocationCallbacks* pAllocator);
```

- `device` is the device which owns `operation`.
- `operation` is the completed operation to be destroyed.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyDeferredOperationKHR-operation-03434
  If `VkAllocationCallbacks` were provided when `operation` was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyDeferredOperationKHR-operation-03435
  If no `VkAllocationCallbacks` were provided when `operation` was created, `pAllocator` must be NULL

- VUID-vkDestroyDeferredOperationKHR-operation-03436
  `operation` must be completed
Valid Usage (Implicit)

- VUID-vkDestroyDeferredOperationKHR-device-parameter
  **device** must be a valid **VkDevice** handle

- VUID-vkDestroyDeferredOperationKHR-operation-parameter
  If **operation** is not **VK_NULL_HANDLE**, **operation** must be a valid **VkDeferredOperationKHR** handle

- VUID-vkDestroyDeferredOperationKHR-pAllocator-parameter
  If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkDestroyDeferredOperationKHR-operation-parent
  If **operation** is a valid handle, it must have been created, allocated, or retrieved from **device**

Host Synchronization

- Host access to **operation** must be externally synchronized

To query the number of additional threads that can usefully be joined to a deferred operation, call:

```c
// Provided by VK_KHR_deferred_host_operations
uint32_t vkGetDeferredOperationMaxConcurrencyKHR(
    VkDevice                      device,
    VkDeferredOperationKHR        operation);
```

- **device** is the device which owns **operation**.
- **operation** is the deferred operation to be queried.

The returned value is the maximum number of threads that can usefully execute a deferred operation concurrently, reported for the state of the deferred operation at the point this command is called. This value is intended to be used to better schedule work onto available threads. Applications can join any number of threads to the deferred operation and expect it to eventually complete, though excessive joins may return **VK_THREAD_DONE_KHR** immediately, performing no useful work.

If **operation** is complete, **vkGetDeferredOperationMaxConcurrencyKHR** returns zero.

If **operation** is currently joined to any threads, the value returned by this command may immediately be out of date.

If **operation** is pending, implementations must not return zero unless at least one thread is currently executing **vkDeferredOperationJoinKHR** on **operation**. If there are such threads, the implementation should return an estimate of the number of additional threads which it could profitably use.
Implementations may return $2^{32}-1$ to indicate that the maximum concurrency is unknown and cannot be easily derived. Implementations may return values larger than the maximum concurrency available on the host CPU. In these situations, an application should clamp the return value rather than oversubscribing the machine.

Note
The recommended usage pattern for applications is to query this value once, after deferral, and schedule no more than the specified number of threads to join the operation. Each time a joined thread receives 
VK_THREAD_IDLE_KHR, the application should schedule an additional join at some point in the future, but is not required to do so.

Valid Usage (Implicit)

- VUID-vkGetDeferredOperationMaxConcurrencyKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetDeferredOperationMaxConcurrencyKHR-operation-parameter
  operation must be a valid VkDeferredOperationKHR handle

- VUID-vkGetDeferredOperationMaxConcurrencyKHR-operation-parent
  operation must have been created, allocated, or retrieved from device

The vkGetDeferredOperationResultKHR function is defined as:

```c
// Provided by VK_KHR_deferred_host_operations
VkResult vkGetDeferredOperationResultKHR(
    VkDevice                                    device,
    VkDeferredOperationKHR                      operation);
```

- device is the device which owns operation.
- operation is the operation whose deferred result is being queried.

If no command has been deferred on operation, vkGetDeferredOperationResultKHR returns VK_SUCCESS.

If the deferred operation is pending, vkGetDeferredOperationResultKHR returns VK_NOT_READY.

If the deferred operation is complete, it returns the appropriate return value from the original command. This value must be one of the VkResult values which could have been returned by the original command if the operation had not been deferred.
Valid Usage (Implicit)

- VUID-vkGetDeferredOperationResultKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkGetDeferredOperationResultKHR-operation-parameter
  operation must be a valid VkDeferredOperationKHR handle

- VUID-vkGetDeferredOperationResultKHR-operation-parent
  operation must have been created, allocated, or retrieved from device

Return Codes

Success
- VK_SUCCESS
- VK_NOT_READY
Chapter 35. Private Data

The private data extension provides a way for users to associate arbitrary user defined data with Vulkan objects. This association is accomplished by storing 64-bit unsigned integers of user defined data in private data slots.

An application can reserve private data slots at device creation. To reserve private data slots, insert a `VkDevicePrivateDataCreateInfoEXT` in the pNext chain in `VkDeviceCreateInfo` before device creation. Multiple `VkDevicePrivateDataCreateInfoEXT` structures can be chained together, and the sum of the requested slots will be reserved. This is an exception to the specified valid usage for structure pointer chains. Reserving slots in this manner is not strictly necessary but it may improve performance.

Private data slots are represented by `VkPrivateDataSlotEXT` handles:

```cpp
// Provided by VK_EXT_private_data
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPrivateDataSlotEXT)
```

To create a private data slot, call:

```cpp
// Provided by VK_EXT_private_data
VkResult vkCreatePrivateDataSlotEXT(
    VkDevice                                    device,
    const VkPrivateDataSlotCreateInfoEXT*       pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkPrivateDataSlotEXT*                       pPrivateDataSlot);
```

- `device` is the logical device associated with the creation of the object(s) holding the private data slot.
- `pCreateInfo` is a pointer to a `VkPrivateDataSlotCreateInfoEXT`
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.
- `pPrivateDataSlot` is a pointer to a `VkPrivateDataSlotEXT` handle in which the resulting private data slot is returned

**Valid Usage**

- `VUID-vkCreatePrivateDataSlotEXT-privateData-04564`
  
  The `privateData` feature must be enabled
Valid Usage (Implicit)

- **VUID-vkCreatePrivateDataSlotEXT-device-parameter**
  
  *device* must be a valid *VkDevice* handle

- **VUID-vkCreatePrivateDataSlotEXT-pCreateInfo-parameter**
  
  *pCreateInfo* must be a valid pointer to a valid *VkPrivateDataSlotCreateInfoEXT* structure

- **VUID-vkCreatePrivateDataSlotEXT-pAllocator-parameter**
  
  If *pAllocator* is not *NULL*, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkCreatePrivateDataSlotEXT-pPrivateDataSlot-parameter**
  
  *pPrivateDataSlot* must be a valid pointer to a *VkPrivateDataSlotEXT* handle

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY

The *VkPrivateDataSlotCreateInfoEXT* structure is defined as:

```c
// Provided by VK_EXT_private_data
typedef struct VkPrivateDataSlotCreateInfoEXT {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPrivateDataSlotCreateFlagsEXT     flags;
} VkPrivateDataSlotCreateInfoEXT;
```

- *sType* is the type of this structure.
- *pNext* is *NULL* or a pointer to a structure extending this structure.
- *flags* is a bitmask of *VkPrivateDataSlotCreateFlagsEXT* specifying additional parameters of the new private data slot
Valid Usage (Implicit)

- **VUID-VkPrivateDataSlotCreateInfoEXT-sType-sType**
  
  
  *sType* must be `VK_STRUCTURE_TYPE_PRIVATE_DATA_SLOT_CREATE_INFO_EXT`

- **VUID-VkPrivateDataSlotCreateInfoEXT-pNext-pNext**
  
  
  *pNext* must be `NULL`

- **VUID-VkPrivateDataSlotCreateInfoEXT-flags-zerobitmask**
  
  
  *flags* must be `0`

// Provided by VK_EXT_private_data

typedef enum VkPrivateDataSlotCreateFlagBitsEXT {
    } VkPrivateDataSlotCreateFlagBitsEXT;

// Provided by VK_EXT_private_data
typedef VkFlags VkPrivateDataSlotCreateFlagsEXT;

VkPrivateDataSlotCreateFlagsEXT is a bitmask type for setting a mask of zero or more VkPrivateDataSlotCreateFlagBitsEXT.

To destroy a private data slot, call:

// Provided by VK_EXT_private_data

void vkDestroyPrivateDataSlotEXT(
    VkDevice device,
    VkPrivateDataSlotEXT privateDataSlot,
    const VkAllocationCallbacks* pAllocator);

- *device* is the logical device associated with the creation of the object(s) holding the private data slot.
- *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.
- *privateDataSlot* is the private data slot to destroy.

Valid Usage

- **VUID-vkDestroyPrivateDataSlotEXT-privateDataSlot-04062**
  
  If *VkAllocationCallbacks* were provided when *privateDataSlot* was created, a compatible set of callbacks must be provided here

- **VUID-vkDestroyPrivateDataSlotEXT-privateDataSlot-04063**
  
  If no *VkAllocationCallbacks* were provided when *privateDataSlot* was created, *pAllocator* must be `NULL`
Valid Usage (Implicit)

- VUID-vkDestroyPrivateDataSlotEXT-device-parameter
  
  **device** must be a valid **VkDevice** handle

- VUID-vkDestroyPrivateDataSlotEXT-privateDataSlot-parameter
  
  If **privateDataSlot** is not **VK_NULL_HANDLE**, **privateDataSlot** must be a valid **VkPrivateDataSlotEXT** handle

- VUID-vkDestroyPrivateDataSlotEXT-pAllocator-parameter
  
  If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure

- VUID-vkDestroyPrivateDataSlotEXT-privateDataSlot-parent
  
  If **privateDataSlot** is a valid handle, it must have been created, allocated, or retrieved from **device**

Host Synchronization

- Host access to **privateDataSlot** must be externally synchronized

To store user defined data in a slot associated with a Vulkan object, call:

```c
// Provided by VK_EXT_private_data
VkResult vkSetPrivateDataEXT(
    VkDevice                                    device,  
    VkObjectType                                objectType, 
    uint64_t                                    objectHandle, 
    VkPrivateDataSlotEXT                        privateDataSlot, 
    uint64_t                                    data);
```

- **device** is the device that created the object.

- **objectType** is a **VkObjectType** specifying the type of object to associate data with.

- **objectHandle** is a handle to the object to associate data with.

- **privateDataSlot** is a handle to a **VkPrivateDataSlotEXT** specifying location of private data storage.

- **data** is user defined data to associate the object with. This data will be stored at **privateDataSlot**.

Valid Usage

- VUID-vkSetPrivateDataEXT-objectHandle-04016
  
  **objectHandle** must be **device** or a child of **device**

- VUID-vkSetPrivateDataEXT-objectHandle-04017
  
  **objectHandle** must be a valid handle to an object of type **objectType**
Valid Usage (Implicit)

- **VUID-vkSetPrivateDataEXT-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkSetPrivateDataEXT-objectType-parameter**
  
  `objectType` must be a valid `VkObjectType` value

- **VUID-vkSetPrivateDataEXT-privateDataSlot-parameter**
  
  `privateDataSlot` must be a valid `VkPrivateDataSlotEXT` handle

- **VUID-vkSetPrivateDataEXT-privateDataSlot-parent**
  
  `privateDataSlot` must have been created, allocated, or retrieved from `device`

Return Codes

**Success**
- `VK_SUCCESS`

**Failure**
- `VK_ERROR_OUT_OF_HOST_MEMORY`

To retrieve user defined data from a slot associated with a Vulkan object, call:

```c
// Provided by VK_EXT_private_data
void vkGetPrivateDataEXT(
    VkDevice device,  // device is the device that created the object
    VkObjectType objectType,  // objectType is a VkObjectType specifying the type of object data is associated with.
    uint64_t objectHandle,  // objectHandle is a handle to the object data is associated with.
    VkPrivateDataSlotEXT privateDataSlot,  // privateDataSlot is a handle to a VkPrivateDataSlotEXT specifying location of private data pointer storage.
    uint64_t* pData);  // pData is a pointer to specify where user data is returned. 0 will be written in the absence of a previous call to vkSetPrivateDataEXT using the object specified by objectHandle.
```

**Note**

Due to platform details on Android, implementations might not be able to reliably return 0 from calls to `vkGetPrivateDataEXT` for `VkSwapchainKHR` objects on which `vkSetPrivateDataEXT` has not previously been called. This erratum is exclusive to the Android platform and objects of type `VkSwapchainKHR`. 
Valid Usage

- VUID-vkGetPrivateDataEXT-objectType-04018
  
  **objectType** must be `VK_OBJECT_TYPE_DEVICE`, or an object type whose parent is `VkDevice`

Valid Usage (Implicit)

- VUID-vkGetPrivateDataEXT-device-parameter
  
  **device** must be a valid `VkDevice` handle

- VUID-vkGetPrivateDataEXT-objectType-parameter
  
  **objectType** must be a valid `VkObjectType` value

- VUID-vkGetPrivateDataEXT-privateDataSlot-parameter
  
  **privateDataSlot** must be a valid `VkPrivateDataSlotEXT` handle

- VUID-vkGetPrivateDataEXT-pData-parameter
  
  **pData** must be a valid pointer to a `uint64_t` value

- VUID-vkGetPrivateDataEXT-privateDataSlot-parent
  
  **privateDataSlot** must have been created, allocated, or retrieved from **device**
Chapter 36. Acceleration Structures

36.1. Acceleration Structures

*Acceleration structures* are data structures used by the implementation to efficiently manage scene geometry as it is *traversed during a ray tracing query*. The application is responsible for managing acceleration structure objects (see *Acceleration Structures*), including allocation, destruction, executing builds or updates, and synchronizing resources used during ray tracing queries.

There are two types of acceleration structures, *top level acceleration structures* and *bottom level acceleration structures*.

![Diagram showing the relationship between top and bottom level acceleration structures](image)

*Figure 24. Acceleration Structure*

**Caption**

The diagram shows the relationship between top and bottom level acceleration structures.

36.1.1. Geometry

*Geometries* refer to a triangle or axis-aligned bounding box.

36.1.2. Top Level Acceleration Structures

Opaque acceleration structure for an array of instances. The descriptor or device address referencing this is the starting point for traversal.

The top level acceleration structure takes a reference to any bottom level acceleration structure referenced by its instances. Those bottom level acceleration structure objects must be valid when the top level acceleration structure is accessed.
36.1.3. Bottom Level Acceleration Structures

Opaque acceleration structure for an array of geometries.

36.1.4. Acceleration Structure Update Rules

The API defines two types of operations to produce acceleration structures from geometry:

- A build operation is used to construct an acceleration structure.
- An update operation is used to modify an existing acceleration structure.

An update operation imposes certain constraints on the input, in exchange for considerably faster execution. When performing an update, the application is required to provide a full description of the acceleration structure, but is prohibited from changing anything other than instance definitions, transform matrices, and vertex or AABB positions. All other aspects of the description must exactly match the one from the original build.

More precisely, the application must not use an update operation to do any of the following:

- Change primitives or instances from active to inactive (as defined in Inactive Primitives and Instances).
- Change the index or vertex formats of triangle geometry.
- Change triangle geometry transform pointers from null to non-null or vice versa.
- Change the number of geometries or instances in the structure.
- Change the geometry flags for any geometry in the structure.
- Change the number of vertices or primitives for any geometry in the structure.

36.1.5. Inactive Primitives and Instances

Acceleration structures allow the use of particular input values to signal inactive primitives or instances.

An inactive triangle is one for which the first (X) component of each vertex is NaN. If any other vertex component is NaN, and the first is not, the behavior is undefined. If the vertex format does not have a NaN representation, then all triangles are considered active.

An inactive instance is one whose acceleration structure handle is VK_NULL_HANDLE.

An inactive AABB is one for which the minimum X coordinate is NaN. If any other component is NaN, and the first is not, the behavior is undefined.

In the above definitions, "NaN" refers to any type of NaN. Signaling, non-signaling, quiet, loud, or otherwise.

An inactive object is considered invisible to all rays, and should not be represented in the acceleration structure. Implementations should ensure that the presence of inactive objects does not seriously degrade traversal performance.
Inactive objects are counted in the auto-generated index sequences which are provided to shaders via `InstanceId` and `PrimitiveId` SPIR-V decorations. This allows objects in the scene to change freely between the active and inactive states, without affecting the layout of any arrays which are being indexed using the ID values.

Any transition between the active and inactive states requires a full acceleration structure rebuild. Applications **must** not perform an acceleration structure update where an object is active in the source acceleration structure but would be inactive in the destination, or vice versa.

### 36.1.6. Building Acceleration Structures

To build an acceleration structure call:

```c
// Provided by VK_NV_ray_tracing
void vkCmdBuildAccelerationStructureNV(
    VkCommandBuffer                             commandBuffer,
    const VkAccelerationStructureInfoNV*        pInfo,
    VkBuffer                                    instanceData,
    VkDeviceSize                                instanceOffset,
    VkBool32                                    update,
    VkAccelerationStructureNV                   dst,
    VkAccelerationStructureNV                   src,
    VkBuffer                                    scratch,
    VkDeviceSize                                scratchOffset);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pInfo` contains the shared information for the acceleration structure's structure.
- `instanceData` is the buffer containing an array of `VkAccelerationStructureInstanceKHR` structures defining acceleration structures. This parameter **must** be `NULL` for bottom level acceleration structures.
- `instanceOffset` is the offset in bytes (relative to the start of `instanceData`) at which the instance data is located.
- `update` specifies whether to update the `dst` acceleration structure with the data in `src`.
- `dst` is a pointer to the target acceleration structure for the build.
- `src` is a pointer to an existing acceleration structure that is to be used to update the `dst` acceleration structure.
- `scratch` is the `VkBuffer` that will be used as scratch memory for the build.
- `scratchOffset` is the offset in bytes relative to the start of `scratch` that will be used as a scratch memory.

Accesses to `dst`, `src`, and `scratch` **must** be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR` or `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR`.
Valid Usage

- **VUID-vkCmdBuildAccelerationStructureNV-geometryCount-02241**
  
  `geometryCount` must be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxGeometryCount`

- **VUID-vkCmdBuildAccelerationStructureNV-dst-02488**
  
  `dst` must have been created with compatible `VkAccelerationStructureInfoNV` where `VkAccelerationStructureInfoNV::type` and `VkAccelerationStructureInfoNV::flags` are identical, `VkAccelerationStructureInfoNV::instanceCount` and `VkAccelerationStructureInfoNV::geometryCount` for `dst` are greater than or equal to the build size and each geometry in `VkAccelerationStructureInfoNV::pGeometries` for `dst` has greater than or equal to the number of vertices, indices, and AABBs.

- **VUID-vkCmdBuildAccelerationStructureNV-update-02489**
  
  If `update` is `VK_TRUE`, `src` must not be `VK_NULL_HANDLE`

- **VUID-vkCmdBuildAccelerationStructureNV-update-02490**
  
  If `update` is `VK_TRUE`, `src` must have been built before with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV` set in `VkAccelerationStructureInfoNV::flags`

- **VUID-vkCmdBuildAccelerationStructureNV-update-02491**
  
  If `update` is `VK_FALSE`, the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `VkAccelerationStructureMemoryRequirementsInfoNV::accelerationStructure` set to `dst` and `VkAccelerationStructureMemoryRequirementsInfoNV::type` set to `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_BUILD_SCRATCH_NV` must be less than or equal to the size of `scratch` minus `scratchOffset`

- **VUID-vkCmdBuildAccelerationStructureNV-update-02492**
  
  If `update` is `VK_TRUE`, the `size` member of the `VkMemoryRequirements` structure returned from a call to `vkGetAccelerationStructureMemoryRequirementsNV` with `VkAccelerationStructureMemoryRequirementsInfoNV::accelerationStructure` set to `dst` and `VkAccelerationStructureMemoryRequirementsInfoNV::type` set to `VK_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_TYPE_UPDATE_SCRATCH_NV` must be less than or equal to the size of `scratch` minus `scratchOffset`

- **VUID-vkCmdBuildAccelerationStructureNV-scratch-03522**
  
  `scratch` must have been created with `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` usage flag

- **VUID-vkCmdBuildAccelerationStructureNV-instanceData-03523**
  
  If `instanceData` is not `VK_NULL_HANDLE`, `instanceData` must have been created with `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` usage flag

- **VUID-vkCmdBuildAccelerationStructureNV-accelerationStructureReference-03786**
  
  Each `VkAccelerationStructureInstanceKHR::accelerationStructureReference` value in `instanceData` must be a valid device address containing a value obtained from `vkGetAccelerationStructureHandleNV`

- **VUID-vkCmdBuildAccelerationStructureNV-update-03524**
  
  If `update` is `VK_TRUE`, then objects that were previously active must not be made inactive as per Inactive Primitives and Instances
If `update` is `VK_TRUE`, then objects that were previously inactive **must** not be made active as per Inactive Primitives and Instances.

If `update` is `VK_TRUE`, the `src` and `dst` objects **must** either be the same object or not have any memory aliasing.

---

**Valid Usage (Implicit)**

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `pInfo` must be a valid pointer to a valid `VkAccelerationStructureInfoNV` structure.
- `instanceData` must be a valid `VkBuffer` handle.
- The `commandBuffer` that was allocated from must support compute operations.
- This command **must** only be called outside of a render pass instance.

---

**Host Synchronization**

- Host access to `commandBuffer` **must** be externally synchronized.
- Host access to the `VkCommandPool` that was allocated from **must** be externally synchronized.
To build acceleration structures call:

```c
// Provided by VK_KHR_acceleration_structure
void vkCmdBuildAccelerationStructuresKHR(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    infoCount,
    const VkAccelerationStructureBuildGeometryInfoKHR* pInfos,
    const VkAccelerationStructureBuildRangeInfoKHR* const* ppBuildRangeInfos);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `infoCount` is the number of acceleration structures to build. It specifies the number of the `pInfos` structures and `ppBuildRangeInfos` pointers that must be provided.
- `pInfos` is an array of `infoCount` `VkAccelerationStructureBuildGeometryInfoKHR` structures defining the geometry used to build each acceleration structure.
- `ppBuildRangeInfos` is an array of `infoCount` pointers to arrays of `VkAccelerationStructureBuildRangeInfoKHR` structures. Each `ppBuildRangeInfos[i]` is an array of `pInfos[i].geometryCount` `VkAccelerationStructureBuildRangeInfoKHR` structures defining dynamic offsets to the addresses where geometry data is stored, as defined by `pInfos[i]`.

The `vkCmdBuildAccelerationStructuresKHR` command provides the ability to initiate multiple acceleration structures builds, however there is no ordering or synchronization implied between any of the individual acceleration structure builds.

**Note**

This means that an application cannot build a top-level acceleration structure in the same `vkCmdBuildAccelerationStructuresKHR` call as the associated bottom-level or instance acceleration structures are being built. There also cannot be any memory aliasing between any acceleration structure memories or scratch memories being used by any of the builds.

Accesses to the acceleration structure scratch buffers as identified by the `VkAccelerationStructureBuildGeometryInfoKHR→scratchData` buffer device addresses must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR` or `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR`. Similarly for accesses to each `VkAccelerationStructureBuildGeometryInfoKHR→srcAccelerationStructure` and `VkAccelerationStructureBuildGeometryInfoKHR→dstAccelerationStructure`. 

---

1886 | Chapter 36. Acceleration Structures
Accesses to other input buffers as identified by any used values of
\texttt{VkAccelerationStructureGeometryTrianglesDataKHR}\rightarrow\texttt{vertexData},
\texttt{VkAccelerationStructureGeometryTrianglesDataKHR}\rightarrow\texttt{indexData},
\texttt{VkAccelerationStructureGeometryTrianglesDataKHR}\rightarrow\texttt{transformData},
\texttt{VkAccelerationStructureGeometryAabbsDataKHR}\rightarrow\texttt{data}, and
\texttt{VkAccelerationStructureGeometryInstancesDataKHR}\rightarrow\texttt{data} \textbf{must} be synchronized with the
\texttt{VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR} pipeline stage and an access type of
\texttt{VK_ACCESS_SHADER_READ_BIT}.
Valid Usage

- VUID-vkCmdBuildAccelerationStructuresKHR-mode-04628
  The mode member of each element of pInfos must be a valid VkBuildAccelerationStructureModeKHR value.

- VUID-vkCmdBuildAccelerationStructuresKHR-srcAccelerationStructure-04629
  If the srcAccelerationStructure member of any element of pInfos is not VK_NULL_HANDLE, the srcAccelerationStructure member must be a valid VkAccelerationStructureKHR handle.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-04630
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, its srcAccelerationStructure member must not be VK_NULL_HANDLE.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03403
  The srcAccelerationStructure member of any element of pInfos must not be the same acceleration structure as the dstAccelerationStructure member of any other element of pInfos.

- VUID-vkCmdBuildAccelerationStructuresKHR-dstAccelerationStructure-03698
  The dstAccelerationStructure member of any element of pInfos must not be the same acceleration structure as the dstAccelerationStructure member of any other element of pInfos.

- VUID-vkCmdBuildAccelerationStructuresKHR-dstAccelerationStructure-03800
  The dstAccelerationStructure member of any element of pInfos must be a valid VkAccelerationStructureKHR handle.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03699
  For each element of pInfos, if its type member is VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR, its dstAccelerationStructure member must have been created with a value of VkAccelerationStructureCreateInfoKHR::type equal to either VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR or VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03700
  For each element of pInfos, if its type member is VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR, its dstAccelerationStructure member must have been created with a value of VkAccelerationStructureCreateInfoKHR::type equal to either VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR or VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03663
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, inactive primitives in its srcAccelerationStructure member must not be made active.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03664
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, active primitives in its srcAccelerationStructure member must not be made inactive.
The `dstAccelerationStructure` member of any element of `pInfos` must not be referenced by the `geometry.instances.data` member of any element of `pGeometries` or `ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR` in any other element of `pInfos`.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `srcAccelerationStructure` member of any other element of `pInfos` with a `mode` equal to `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, which is accessed by this command.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `scratchData` member of any element of `pInfos` (including the same element), which is accessed by this command.

The range of memory backing the `scratchData` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `srcAccelerationStructure` member of any other element of `pInfos` with a `mode` equal to `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR` (including the same element), which is accessed by this command.

The range of memory backing the `scratchData` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing any acceleration structure referenced by the `geometry.instances.data` member of any element of `pGeometries` or `ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR` in any other element of `pInfos`, which is accessed by this command.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member must not be `VK_NULL_HANDLE`.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member must have been built before with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR` set in `VkAccelerationStructureBuildGeometryInfoKHR::flags`.
For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` and `dstAccelerationStructure` members must either be the same `VkAccelerationStructureKHR`, or not have any memory aliasing.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `geometryCount` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `flags` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `type` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, its `geometryType` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, its `flags` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, if `geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, its `geometry.triangles.vertexFormat` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, if `geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, its `geometry.triangles.maxVertex` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is
VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its
gometry.triangles.indexType member must have the same value which was specified
when srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03766
  For each element of pInfos, if its mode member is
  VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
  VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
  ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its
gometry.triangles.transformData member must have the same value which was specified
  when srcAccelerationStructure was last built, then it must be NULL.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03767
  For each element of pInfos, if its mode member is
  VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
  VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
  ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, if its
gometry.triangles.transformData member was NULL when srcAccelerationStructure
  was last built, then it may not be NULL.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03768
  For each element of pInfos, if its mode member is
  VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
  VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
  ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, and
  geometry.triangles.indexType is not VK_INDEX_TYPE_NONE_KHR, then the value of each index
  referenced index must be the same as the corresponding index value when srcAccelerationStructure
  was last built.

- VUID-vkCmdBuildAccelerationStructuresKHR-primitiveCount-03769
  For each VkAccelerationStructureBuildRangeInfoKHR referenced by this command, its
  primitiveCount member must have the same value which was specified when
  srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresKHR-firstVertex-03770
  For each VkAccelerationStructureBuildRangeInfoKHR referenced by this command, if the
  corresponding geometry uses indices, its firstVertex member must have the same value
  which was specified when srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03801
  For each element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of
  VK_GEOMETRY_TYPE_INSTANCES_KHR, the corresponding ppBuildRangeInfos[i][j].primitiveCount
  must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR
  ::maxInstanceCount

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03707
  For each element of pInfos, the buffer used to create its dstAccelerationStructure member
  must be bound to device memory

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03708
  

Chapter 36. Acceleration Structures | 1891
For each element of \texttt{pInfos}, if its \texttt{mode} member is \texttt{VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR} the buffer used to create its \texttt{srcAccelerationStructure} member must be bound to device memory.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03709

For each element of \texttt{pInfos}, the buffer used to create each acceleration structure referenced by the \texttt{geometry.instances.data} member of any element of \texttt{pGeometries} or \texttt{ppGeometries} with a \texttt{geometryType} of \texttt{VK_GEOMETRY_TYPE_INSTANCES_KHR} must be bound to device memory.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03671

If \texttt{pInfos[i].mode} is \texttt{VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR}, all addresses between \texttt{pInfos[i].scratchData.deviceAddress} and \texttt{pInfos[i].scratchData.deviceAddress + N - 1} must be in the buffer device address range of the same buffer, where \texttt{N} is given by the \texttt{buildScratchSize} member of the \texttt{VkAccelerationStructureBuildSizesInfoKHR} structure returned from a call to \texttt{vkGetAccelerationStructureBuildSizesKHR} with an identical \texttt{VkAccelerationStructureBuildGeometryInfoKHR} structure and primitive count.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03672

If \texttt{pInfos[i].mode} is \texttt{VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR}, all addresses between \texttt{pInfos[i].scratchData.deviceAddress} and \texttt{pInfos[i].scratchData.deviceAddress + N - 1} must be in the buffer device address range of the same buffer, where \texttt{N} is given by the \texttt{updateScratchSize} member of the \texttt{VkAccelerationStructureBuildSizesInfoKHR} structure returned from a call to \texttt{vkGetAccelerationStructureBuildSizesKHR} with an identical \texttt{VkAccelerationStructureBuildGeometryInfoKHR} structure and primitive count.

- VUID-vkCmdBuildAccelerationStructuresKHR-geometry-03673

The buffers from which the buffer device addresses for all of the \texttt{geometry.triangles.vertexData}, \texttt{geometry.triangles.indexData}, \texttt{geometry.triangles.transformData}, \texttt{geometry.aabbs.data}, and \texttt{geometry.instances.data} members of all \texttt{pInfos[i].pGeometries} and \texttt{pInfos[i].ppGeometries} must have been created with the \texttt{VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR} usage flag.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03674

The buffer from which the buffer device address \texttt{pInfos[i].scratchData.deviceAddress} is queried must have been created with \texttt{VK_BUFFER_USAGE_STORAGE_BUFFER_BIT} usage flag.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03802

For each element of \texttt{pInfos}, its \texttt{scratchData.deviceAddress} member must be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03803

For each element of \texttt{pInfos}, if \texttt{scratchData.deviceAddress} is the address of a non-sparse buffer then it must be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03710

For each element of \texttt{pInfos}, its \texttt{scratchData.deviceAddress} member must be a multiple of \texttt{VkPhysicalDeviceAccelerationStructurePropertiesKHR::minAccelerationStructureScratchOffsetAlignment}.

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03804
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, \( \mathbf{geometry.triangles.vertexData.deviceAddress} \) \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03805
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, if \( \mathbf{geometry.triangles.vertexData.deviceAddress} \) is the address of a non-sparse buffer then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03711
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, \( \mathbf{geometry.triangles.vertexData.deviceAddress} \) \textbf{must} be aligned to the size in bytes of the smallest component of the format in \texttt{vertexFormat}

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03806
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, if \( \mathbf{geometry.triangles.indexType} \) is not \texttt{VK\_INDEX\_TYPE\_NONE\_KHR}, \( \mathbf{geometry.triangles.indexData.deviceAddress} \) \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03807
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, if \( \mathbf{geometry.triangles.indexData.deviceAddress} \) is the address of a non-sparse buffer then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03810
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, if \( \mathbf{geometry.triangles.transformData.deviceAddress} \) is not \texttt{0}, it \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}

- VUID-vkCmdBuildAccelerationStructuresKHR-pInfos-03811
For any element of \( \mathbf{pInfos[i].pGeometries} \) or \( \mathbf{pInfos[i].ppGeometries} \) with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\__TRIANGLES\_KHR}, if \( \mathbf{geometry.triangles.transformData.deviceAddress} \) is not \texttt{0}, it \textbf{must} be aligned to 16 bytes
For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_AABBS_KHR`, if `geometry.aabbs.data.deviceAddress` is the address of a non-sparse buffer then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_AABBS_KHR`, `geometry.aabbs.data.deviceAddress` **must** be aligned to 8 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.arrayOfPointers` is `VK_FALSE`, `geometry.instances.data.deviceAddress` **must** be aligned to 16 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.arrayOfPointers` is `VK_TRUE`, each element of `geometry.instances.data.deviceAddress` in device memory **must** be aligned to 8 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, each `VkAccelerationStructureInstanceKHR`::`accelerationStructureReference` value in `geometry.instances.data.deviceAddress` **must** be a valid device address containing a value obtained from `vkGetAccelerationStructureDeviceAddressKHR`.

For each `pInfos[i]`, `dstAccelerationStructure` **must** have been created with a value of `VkAccelerationStructureCreateInfoKHR`::`size` greater than or equal to the memory size required by the build operation, as returned by `vkGetAccelerationStructureBuildSizesKHR` with `pBuildInfo = pInfos[i]` and with each element of the `pMaxPrimitiveCounts` array greater than or equal to the equivalent `ppBuildRangeInfos[i][j].primitiveCount` values for `j` in `[0, pInfos[i].geometryCount)`. 

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.instances.data.deviceAddress` is the address of a non-sparse buffer then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.instances.data.deviceAddress` **must** be aligned to 8 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, each `VkAccelerationStructureInstanceKHR`::`accelerationStructureReference` value in `geometry.instances.data.deviceAddress` **must** be a valid device address containing a value obtained from `vkGetAccelerationStructureDeviceAddressKHR`.
Each element of `ppBuildRangeInfos[i]` **must** be a valid pointer to an array of `pInfos[i].geometryCount` `VkAccelerationStructureBuildRangeInfoKHR` structures.

### Valid Usage (Implicit)

- **commandBuffer** must be a valid `VkCommandBuffer` handle
- `pInfos` must be a valid pointer to an array of `infoCount` valid `VkAccelerationStructureBuildGeometryInfoKHR` structures
- `ppBuildRangeInfos` must be a valid pointer to an array of `infoCount` `VkAccelerationStructureBuildRangeInfoKHR` structures
- `commandBuffer` must be in the **recording** state
- The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations
- This command **must** only be called outside of a render pass instance
- `infoCount` must be greater than 0

### Host Synchronization

- Host access to `commandBuffer` **must** be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from **must** be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Secondary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To build acceleration structures with some parameters sourced on the device call:
// Provided by VK_KHR_acceleration_structure
void vkCmdBuildAccelerationStructuresIndirectKHR(
    VkCommandBuffer commandBuffer,
    uint32_t infoCount,
    const VkAccelerationStructureBuildGeometryInfoKHR* pInfos,
    const VkDeviceAddress* pIndirectDeviceAddresses,
    const uint32_t* pIndirectStrides,
    const uint32_t* const* ppMaxPrimitiveCounts);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `infoCount` is the number of acceleration structures to build.
- `pInfos` is an array of `infoCount` `VkAccelerationStructureBuildGeometryInfoKHR` structures defining the geometry used to build each acceleration structure.
- `pIndirectDeviceAddresses` is an array of `infoCount` buffer device addresses which point to `pInfos[i]`–`geometryCount` `VkAccelerationStructureBuildRangeInfoKHR` structures defining dynamic offsets to the addresses where geometry data is stored, as defined by `pInfos[i]`.
- `pIndirectStrides` is an array of `infoCount` byte strides between elements of `pIndirectDeviceAddresses`.
- `ppMaxPrimitiveCounts` is an array of `infoCount` arrays of `pInfo[i]`–`geometryCount` values indicating the maximum number of primitives that will be built by this command for each geometry.

Accesses to acceleration structures, scratch buffers, vertex buffers, index buffers, and instance buffers must be synchronized as with `vkCmdBuildAccelerationStructuresKHR`.

Accesses to any element of `pIndirectDeviceAddresses` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_INDIRECT_COMMAND_READ_BIT`. 
Valid Usage

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-mode-04628**
  The *mode* member of each element of *pInfos* must be a valid *VkBuildAccelerationStructureModeKHR* value

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-srcAccelerationStructure-04629**
  If the *srcAccelerationStructure* member of any element of *pInfos* is not *VK_NULL_HANDLE*, the *srcAccelerationStructure* member must be a valid *VkAccelerationStructureKHR* handle

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-04630**
  For each element of *pInfos*, if its *mode* member is *VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR*, its *srcAccelerationStructure* member must not be *VK_NULL_HANDLE*

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03403**
  The *srcAccelerationStructure* member of any element of *pInfos* must not be the same acceleration structure as the *dstAccelerationStructure* member of any other element of *pInfos*

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-dstAccelerationStructure-03698**
  The *dstAccelerationStructure* member of any element of *pInfos* must not be the same acceleration structure as the *dstAccelerationStructure* member of any other element of *pInfos*

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-dstAccelerationStructure-03800**
  The *dstAccelerationStructure* member of any element of *pInfos* must be a valid *VkAccelerationStructureKHR* handle

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03699**
  For each element of *pInfos*, if its *type* member is *VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR*, its *dstAccelerationStructure* member must have been created with a value of *VkAccelerationStructureCreateInfoKHR::type* equal to either *VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR* or *
  VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR*

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03700**
  For each element of *pInfos*, if its *type* member is *VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR*, its *dstAccelerationStructure* member must have been created with a value of *VkAccelerationStructureCreateInfoKHR::type* equal to either *VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR* or *
  VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR*

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03663**
  For each element of *pInfos*, if its *mode* member is *VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR*, inactive primitives in its *srcAccelerationStructure* member must not be made active

- **VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03664**
  For each element of *pInfos*, if its *mode* member is *VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR*, active primitives in its *srcAccelerationStructure* member must not be made inactive
The `dstAccelerationStructure` member of any element of `pInfos` must not be referenced by the `geometry.instances.data` member of any element of `pGeometries` or `ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR` in any other element of `pInfos`.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `srcAccelerationStructure` member of any other element of `pInfos` with a `mode` equal to `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, which is accessed by this command.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `dstAccelerationStructure` member of any other element of `pInfos`, which is accessed by this command.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `scratchData` member of any element of `pInfos` (including the same element), which is accessed by this command.

The range of memory backing the `scratchData` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing the `scratchData` member of any other element of `pInfos`, which is accessed by this command.

The range of memory backing the `scratchData` member of any element of `pInfos` that is accessed by this command must not overlap the memory backing any acceleration structure referenced by the `geometry.instances.data` member of any element of `pGeometries` or `ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR` in any other element of `pInfos`, which is accessed by this command.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member must not be `VK_NULL_HANDLE`.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member must have been built before with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR` set in `VkAccelerationStructureBuildGeometryInfoKHR::flags`.
For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` and `dstAccelerationStructure` members must either be the same `VkAccelerationStructureKHR`, or not have any memory aliasing.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `geometryCount` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `flags` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, its `geometryType` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, if `geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, its `geometry.triangles.vertexFormat` member must have the same value which was specified when `srcAccelerationStructure` was last built.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, then for each `VkAccelerationStructureGeometryKHR` structure referred to by its `pGeometries` or `ppGeometries` members, if `geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, its `geometry.triangles.maxVertex` member must have the same value which was specified when `srcAccelerationStructure` was last built.
VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its geometry.triangles.indexType member must have the same value which was specified when srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03766
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its geometry.triangles.transformData member must be NULL when srcAccelerationStructure was last built, then it must be NULL.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03767
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, if its geometry.triangles.transformData member was NULL when srcAccelerationStructure was last built, then it may not be NULL.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03768
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, and geometry.triangles.indexType is not VK_INDEX_TYPE_NONE_KHR, then the value of each index referenced index must be the same as the corresponding index value when srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-primitiveCount-03769
  For each VkAccelerationStructureBuildRangeInfoKHR referenced by this command, its primitiveCount member must have the same value which was specified when srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-firstVertex-03770
  For each VkAccelerationStructureBuildRangeInfoKHR referenced by this command, if the corresponding geometry uses indices, its firstVertex member must have the same value which was specified when srcAccelerationStructure was last built.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03801
  For each element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_INSTANCES_KHR, the corresponding ppMaxPrimitiveCounts[i][j] must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxInstanceCount

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03708
  For each element of pInfos, the buffer used to create its dstAccelerationStructure member must be bound to device memory

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03708
  For each element of pInfos, if its mode member is
VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR the buffer used to create its srcAccelerationStructure member must be bound to device memory

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03709
  For each element of pInfos, the buffer used to create each acceleration structure referenced by the geometry.instances.data member of any element of pGeometries or ppGeometries with a geometryType of VK_GEOMETRY_TYPE_INSTANCES_KHR must be bound to device memory

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03671
  If pInfos[i].mode is VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR, all addresses between pInfos[i].scratchData.deviceAddress and pInfos[i].scratchData.deviceAddress + N - 1 must be in the buffer device address range of the same buffer, where N is given by the buildScratchSize member of the VkAccelerationStructureBuildSizesInfoKHR structure returned from a call to vkGetAccelerationStructureBuildSizesKHR with an identical VkAccelerationStructureBuildGeometryInfoKHR structure and primitive count

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03672
  If pInfos[i].mode is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, all addresses between pInfos[i].scratchData.deviceAddress and pInfos[i].scratchData.deviceAddress + N - 1 must be in the buffer device address range of the same buffer, where N is given by the updateScratchSize member of the VkAccelerationStructureBuildSizesInfoKHR structure returned from a call to vkGetAccelerationStructureBuildSizesKHR with an identical VkAccelerationStructureBuildGeometryInfoKHR structure and primitive count

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03673
  The buffers from which the buffer device addresses for all of the geometry.triangles.vertexData, geometry.triangles.indexData, geometry.triangles.transformData, geometry.aabbs.data, and geometry.instances.data members of all pInfos[i].pGeometries and pInfos[i].ppGeometries must have been created with the VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR usage flag

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03674
  The buffer from which the buffer device address pInfos[i].scratchData.deviceAddress is queried must have been created with VK_BUFFER_USAGE_STORAGE_BUFFER_BIT usage flag

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03802
  For each element of pInfos, its scratchData.deviceAddress member must be a valid device address obtained from vkGetBufferDeviceAddress

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03803
  For each element of pInfos, if scratchData.deviceAddress is the address of a non-sparse buffer then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03710
  For each element of pInfos, its scratchData.deviceAddress member must be a multiple of VkPhysicalDeviceAccelerationStructurePropertiesKHR::minAccelerationStructureScratchOffsetAlignment

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03804
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of
\texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, \texttt{geometry.triangles.vertexData.deviceAddress} \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03805}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, if \texttt{geometry.triangles.vertexData.deviceAddress} is the address of a non-sparse buffer then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03711}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, \texttt{geometry.triangles.vertexData.deviceAddress} \textbf{must} be aligned to the size in bytes of the smallest component of the format in \texttt{vertexFormat}

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03806}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, if \texttt{geometry.triangles.indexData.deviceAddress} is not \texttt{VK\_INDEX\_TYPE\_NONE\_KHR}, \texttt{geometry.triangles.indexData.deviceAddress} \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03807}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, if \texttt{geometry.triangles.indexType} is not \texttt{VK\_INDEX\_TYPE\_NONE\_KHR}, \texttt{geometry.triangles.indexType} not equal to \texttt{VK\_INDEX\_TYPE\_NONE\_KHR}, \texttt{geometry.triangles.indexData.deviceAddress} \textbf{must} be aligned to the size in bytes of the type in \texttt{indexType}

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03808}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, if \texttt{geometry.triangles.transformData.deviceAddress} is not 0, it \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03809}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, if \texttt{geometry.triangles.transformData.deviceAddress} is the address of a non-sparse buffer then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03810}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_KHR}, if \texttt{geometry.triangles.transformData.deviceAddress} is not 0, it \textbf{must} be aligned to 16 bytes

- \textbf{VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03811}
  
  For any element of \texttt{pInfos[i].pGeometries} or \texttt{pInfos[i].ppGeometries} with a \texttt{geometryType} of \texttt{VK\_GEOMETRY\_TYPE\_AABBS\_KHR}, \texttt{geometry.aabbs.data.deviceAddress} \textbf{must} be a valid device address obtained from \texttt{vkGetBufferDeviceAddress}
For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_AABBS_KHR`, if `geometry.aabbs.data.deviceAddress` is the address of a non-sparse buffer then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_AABBS_KHR`, `geometry.aabbs.data.deviceAddress` **must** be aligned to 8 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.arrayOfPointers` is `VK_FALSE`, `geometry.instances.data.deviceAddress` **must** be aligned to 16 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.arrayOfPointers` is `VK_TRUE`, each element of `geometry.instances.data.deviceAddress` in device memory **must** be aligned to 8 bytes.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, if `geometry.instances.data.deviceAddress` is the address of a non-sparse buffer then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

For any element of `pInfos[i].pGeometries` or `pInfos[i].ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR`, each `VkAccelerationStructureInstanceKHR::accelerationStructureReference` value in `geometry.instances.data.deviceAddress` **must** be a valid device address containing a value obtained from `vkGetAccelerationStructureDeviceAddressKHR`.

For any element of `pIndirectDeviceAddresses`, if the buffer from which it was queried is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

For any element of `pIndirectDeviceAddresses[i]`, all device addresses between `pIndirectDeviceAddresses[i]` and `pIndirectDeviceAddresses[i] + (pInfos[i].geometryCount × pIndirectStrides[i]) - 1` **must** be in the buffer device address range of the same buffer.
For any element of \texttt{pIndirectDeviceAddresses}, the buffer from which it was queried must have been created with the \texttt{VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT} bit set.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pIndirectDeviceAddresses-03648 Each element of \texttt{pIndirectDeviceAddresses} must be a multiple of 4.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pIndirectDeviceAddresses-03787 Each element of \texttt{pIndirectDeviceAddresses} must be a multiple of 4.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-commandBuffer-03649 \texttt{commandBuffer} must not be a protected command buffer.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-accelerationStructureIndirectBuild-03650 The \texttt{VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureIndirectBuild} feature must be enabled.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pIndirectDeviceAddresses-03651 Each \texttt{VkAccelerationStructureBuildRangeInfoKHR} structure referenced by any element of \texttt{pIndirectDeviceAddresses} must be a valid \texttt{VkAccelerationStructureBuildRangeInfoKHR} structure.

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-03652 \texttt{pInfos[i].dstAccelerationStructure} must have been created with a value of \texttt{VkAccelerationStructureCreateInfoKHR::size} greater than or equal to the memory size required by the build operation, as returned by \texttt{vkGetAccelerationStructureBuildSizesKHR} with \texttt{pBuildInfo = pInfos[i]} and \texttt{pMaxPrimitiveCounts = ppMaxPrimitiveCounts[i]}

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-ppMaxPrimitiveCounts-03653 Each \texttt{ppMaxPrimitiveCounts[i][j]} must be greater than or equal to the the \texttt{primitiveCount} value specified by the \texttt{VkAccelerationStructureBuildRangeInfoKHR} structure located at \texttt{pIndirectDeviceAddresses[i] + (j \times pIndirectStrides[i])}
Valid Usage (Implicit)

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pInfos-parameter
  pInfos must be a valid pointer to an array of infoCount valid
  VkAccelerationStructureBuildGeometryInfoKHR structures

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pIndirectDeviceAddresses-parameter
  pIndirectDeviceAddresses must be a valid pointer to an array of infoCount VkDeviceAddress
  values

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-pIndirectStrides-parameter
  pIndirectStrides must be a valid pointer to an array of infoCount uint32_t values

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-ppMaxPrimitiveCounts-parameter
  ppMaxPrimitiveCounts must be a valid pointer to an array of infoCount uint32_t values

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support compute
  operations

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdBuildAccelerationStructuresIndirectKHR-infoCount-arraylength
  infoCount must be greater than 0

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally
  synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The VkAccelerationStructureBuildGeometryInfoKHR structure is defined as:
// Provided by VK_KHR_acceleration_structure

typedef struct VkAccelerationStructureBuildGeometryInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureTypeKHR type;
    VkBuildAccelerationStructureFlagsKHR flags;
    VkBuildAccelerationStructureModeKHR mode;
    VkAccelerationStructureKHR srcAccelerationStructure;
    VkAccelerationStructureKHR dstAccelerationStructure;
    uint32_t geometryCount;
    const VkAccelerationStructureGeometryKHR* pGeometries;
    const VkAccelerationStructureGeometryKHR* const* ppGeometries;
    VkDeviceOrHostAddressKHR scratchData;
} VkAccelerationStructureBuildGeometryInfoKHR;

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *type* is a `VkAccelerationStructureTypeKHR` value specifying the type of acceleration structure being built.
- *flags* is a bitmask of `VkBuildAccelerationStructureFlagBitsKHR` specifying additional parameters of the acceleration structure.
- *mode* is a `VkBuildAccelerationStructureModeKHR` value specifying the type of operation to perform.
- *srcAccelerationStructure* points to an existing acceleration structure that is to be used to update the *dst* acceleration structure when *mode* is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`.
- *dstAccelerationStructure* points to the target acceleration structure for the build.
- *geometryCount* specifies the number of geometries that will be built into *dstAccelerationStructure*.
- *pGeometries* is a pointer to an array of `VkAccelerationStructureGeometryKHR` structures.
- *ppGeometries* is a pointer to an array of pointers to `VkAccelerationStructureGeometryKHR` structures.
- *scratchData* is the device or host address to memory that will be used as scratch memory for the build.

Only one of *pGeometries* or *ppGeometries* can be a valid pointer, the other must be NULL. Each element of the non-NULL array describes the data used to build each acceleration structure geometry.

The index of each element of the *pGeometries* or *ppGeometries* members of `VkAccelerationStructureBuildGeometryInfoKHR` is used as the geometry index during ray traversal. The geometry index is available in ray shaders via the `RayGeometryIndexKHR` built-in, and is used to determine hit and intersection shaders executed during traversal. The geometry index is available to ray queries via the `OpRayQueryGetIntersectionGeometryIndexKHR` instruction.
Valid Usage

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03654
  
  type must not be VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-pGeometries-03788
  
  Only one of pGeometries or ppGeometries can be a valid pointer, the other must be NULL

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03789
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR, the geometryType member of
  elements of either pGeometries or ppGeometries must be VK_GEOMETRY_TYPE_INSTANCES_KHR

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03790
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR, geometryCount must be 1

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03791
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR the geometryType member of
  elements of either pGeometries or ppGeometries must not be
  VK_GEOMETRY_TYPE_INSTANCES_KHR

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03792
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR then the geometryType member
  of each geometry in either pGeometries or ppGeometries must be the same

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03793
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR then geometryCount must be
  less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR
  ::maxGeometryCount

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03794
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR and the geometryType member
  of either pGeometries or ppGeometries is VK_GEOMETRY_TYPE_AABBS_KHR, the total number of
  AABBs in all geometries must be less than or equal to
  VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxPrimitiveCount

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-03795
  
  If type is VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR and the geometryType member
  of either pGeometries or ppGeometries is VK_GEOMETRY_TYPE_TRIANGLES_KHR, the total number
  of triangles in all geometries must be less than or equal to
  VkPhysicalDeviceAccelerationStructurePropertiesKHR::maxPrimitiveCount

- VUID-VkAccelerationStructureBuildGeometryInfoKHR-flags-03796
  
  If flags has the VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR bit set, then it
  must not have the VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_KHR bit set
Valid Usage (Implicit)

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR`

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-pNext-pNext**
  - `pNext` must be `NULL`

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-type-parameter**
  - `type` must be a valid `VkAccelerationStructureTypeKHR` value

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-flags-parameter**
  - `flags` must be a valid combination of `VkBuildAccelerationStructureFlagBitsKHR` values

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-pGeometries-parameter**
  - If `geometryCount` is not 0, and `pGeometries` is not `NULL`, `pGeometries` must be a valid pointer to an array of `geometryCount` valid `VkAccelerationStructureGeometryKHR` structures

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-ppGeometries-parameter**
  - If `geometryCount` is not 0, and `ppGeometries` is not `NULL`, `ppGeometries` must be a valid pointer to an array of `geometryCount` valid pointers to valid `VkAccelerationStructureGeometryKHR` structures

- **VUID-VkAccelerationStructureBuildGeometryInfoKHR-commonparent**
  - Both of `dstAccelerationStructure`, and `srcAccelerationStructure` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

The `VkBuildAccelerationStructureModeKHR` enumeration is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef enum VkBuildAccelerationStructureModeKHR {
    VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR = 0,
    VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR = 1,
} VkBuildAccelerationStructureModeKHR;
```

- **VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR** specifies that the destination acceleration structure will be built using the specified geometries.

- **VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR** specifies that the destination acceleration structure will be built using data in a source acceleration structure, updated by the specified geometries.

The `VkDeviceOrHostAddressKHR` union is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef union VkDeviceOrHostAddressKHR {
    VkDeviceAddress    deviceAddress;
    void*              hostAddress;
} VkDeviceOrHostAddressKHR;
```
• **deviceAddress** is a buffer device address as returned by the `vkGetBufferDeviceAddressKHR` command.

• **hostAddress** is a host memory address.

The `VkDeviceOrHostAddressConstKHR` union is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef union VkDeviceOrHostAddressConstKHR {
  VkDeviceAddress deviceAddress;
  const void* hostAddress;
} VkDeviceOrHostAddressConstKHR;
```

• **deviceAddress** is a buffer device address as returned by the `vkGetBufferDeviceAddressKHR` command.

• **hostAddress** is a const host memory address.

The `VkAccelerationStructureGeometryKHR` structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureGeometryKHR {
  VkStructureType sType;
  const void* pNext;
  VkGeometryTypeKHR geometryType;
  VkAccelerationStructureGeometryDataKHR geometry;
  VkGeometryFlagsKHR flags;
} VkAccelerationStructureGeometryKHR;
```

• **sType** is the type of this structure.

• **pNext** is `NULL` or a pointer to a structure extending this structure.

• **geometryType** describes which type of geometry this `VkAccelerationStructureGeometryKHR` refers to.

• **geometry** is a `VkAccelerationStructureGeometryDataKHR` union describing the geometry data for the relevant geometry type.

• **flags** is a bitmask of `VkGeometryFlagBitsKHR` values describing additional properties of how the geometry should be built.
Valid Usage

- **VUID-VkAccelerationStructureGeometryKHR-geometryType-03541**
  If `geometryType` is `VK_GEOMETRY_TYPE_AABBS_KHR`, the `aabbs` member of `geometry` must be a valid `VkAccelerationStructureGeometryAabbsDataKHR` structure.

- **VUID-VkAccelerationStructureGeometryKHR-geometryType-03542**
  If `geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, the `triangles` member of `geometry` must be a valid `VkAccelerationStructureGeometryTrianglesDataKHR` structure.

- **VUID-VkAccelerationStructureGeometryKHR-geometryType-03543**
  If `geometryType` is `VK_GEOMETRY_TYPE_INSTANCES_KHR`, the `instances` member of `geometry` must be a valid `VkAccelerationStructureGeometryInstancesDataKHR` structure.

Valid Usage (Implicit)

- **VUID-VkAccelerationStructureGeometryKHR-sType-sType**
  `sType` must be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR`.

- **VUID-VkAccelerationStructureGeometryKHR-pNext-pNext**
  `pNext` must be `NULL`.

- **VUID-VkAccelerationStructureGeometryKHR-geometryType-parameter**
  `geometryType` must be a valid `VkGeometryTypeKHR` value.

- **VUID-VkAccelerationStructureGeometryKHR-triangles-parameter**
  If `geometryType` is `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, the `triangles` member of `geometry` must be a valid `VkAccelerationStructureGeometryTrianglesDataKHR` structure.

- **VUID-VkAccelerationStructureGeometryKHR-aabbs-parameter**
  If `geometryType` is `VK_GEOMETRY_TYPE_AABBS_KHR`, the `aabbs` member of `geometry` must be a valid `VkAccelerationStructureGeometryAabbsDataKHR` structure.

- **VUID-VkAccelerationStructureGeometryKHR-instances-parameter**
  If `geometryType` is `VK_GEOMETRY_TYPE_INSTANCES_KHR`, the `instances` member of `geometry` must be a valid `VkAccelerationStructureGeometryInstancesDataKHR` structure.

- **VUID-VkAccelerationStructureGeometryKHR-flags-parameter**
  `flags` must be a valid combination of `VkGeometryFlagBitsKHR` values.

The `VkAccelerationStructureGeometryDataKHR` union is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef union VkAccelerationStructureGeometryDataKHR {
    VkAccelerationStructureGeometryTrianglesDataKHR    triangles;
    VkAccelerationStructureGeometryAabbsDataKHR        aabbs;
    VkAccelerationStructureGeometryInstancesDataKHR    instances;
} VkAccelerationStructureGeometryDataKHR;
```

- `triangles` is a `VkAccelerationStructureGeometryTrianglesDataKHR` structure.
• *aabb* is a *VkAccelerationStructureGeometryAabbsDataKHR* structure.
• *instances* is a *VkAccelerationStructureGeometryInstancesDataKHR* structure.

The *VkAccelerationStructureGeometryTrianglesDataKHR* structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureGeometryTrianglesDataKHR {
    VkStructureType sType;
    const void* pNext;
    VkFormat vertexFormat;
    VkDeviceOrHostAddressConstKHR vertexData;
    VkDeviceSize vertexStride;
    uint32_t maxVertex;
    VkIndexType indexType;
    VkDeviceOrHostAddressConstKHR indexData;
    VkDeviceOrHostAddressConstKHR transformData;
} VkAccelerationStructureGeometryTrianglesDataKHR;
```

- *sType* is the type of this structure.
- *pNext* is *NULL* or a pointer to a structure extending this structure.
- *vertexFormat* is the *VkFormat* of each vertex element.
- *vertexData* is a device or host address to memory containing vertex data for this geometry.
- *maxVertex* is the highest index of a vertex that will be addressed by a build command using this structure.
- *vertexStride* is the stride in bytes between each vertex.
- *indexType* is the *VkIndexType* of each index element.
- *indexData* is a device or host address to memory containing index data for this geometry.
- *transformData* is a device or host address to memory containing an optional reference to a *VkTransformMatrixKHR* structure defining a transformation that should be applied to vertices in this geometry.

*Note*

Unlike the stride for vertex buffers in *VkVertexInputBindingDescription* for graphics pipelines which must not exceed *maxVertexInputBindingStride*, *vertexStride* for acceleration structure geometry is instead restricted to being a 32-bit value.
Valid Usage

- `vertexStride` must be a multiple of the size in bytes of the smallest component of `vertexFormat`
- `vertexStride` must be less than or equal to $2^{32} - 1$
- `vertexFormat` must support the `VK_FORMAT_FEATURE ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR` in `VkFormatProperties::bufferFeatures` as returned by `vkGetPhysicalDeviceFormatProperties2`
- `indexType` must be `VK_INDEX_TYPE_UINT16`, `VK_INDEX_TYPE_UINT32`, or `VK_INDEX_TYPE_NONE_KHR`

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR`
- `pNext` must be `NULL`
- `vertexFormat` must be a valid `VkFormat` value
- `indexType` must be a valid `VkIndexType` value

The `VkTransformMatrixKHR` structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkTransformMatrixKHR {
    float matrix[3][4];
} VkTransformMatrixKHR;
```

or the equivalent

```c
// Provided by VK_NV_ray_tracing
typedef VkTransformMatrixKHR VkTransformMatrixNV;
```

- `matrix` is a 3x4 row-major affine transformation matrix.
Valid Usage

- VUID-VkTransformMatrixKHR-matrix-03799
  The first three columns of matrix must define an invertible 3x3 matrix

The VkAccelerationStructureGeometryAabbsDataKHR structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureGeometryAabbsDataKHR {
    VkStructureType                  sType;
    const void*                      pNext;
    VkDeviceOrHostAddressConstKHR    data;
    VkDeviceSize                     stride;
} VkAccelerationStructureGeometryAabbsDataKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **data** is a device or host address to memory containing VkAabbPositionsKHR structures containing position data for each axis-aligned bounding box in the geometry.
- **stride** is the stride in bytes between each entry in data. The stride must be a multiple of 8.

Valid Usage

- VUID-VkAccelerationStructureGeometryAabbsDataKHR-stride-03545
  stride must be a multiple of 8
- VUID-VkAccelerationStructureGeometryAabbsDataKHR-stride-03820
  stride must be less than or equal to $2^{32} - 1$

Valid Usage (Implicit)

- VUID-VkAccelerationStructureGeometryAabbsDataKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_AABBS_DATA_KHR
- VUID-VkAccelerationStructureGeometryAabbsDataKHR-pNext-pNext
  pNext must be NULL

The VkAabbPositionsKHR structure is defined as:
typedef struct VkAabbPositionsKHR {
    float minX;
    float minY;
    float minZ;
    float maxX;
    float maxY;
    float maxZ;
} VkAabbPositionsKHR;

or the equivalent

typeof VkAabbPositionsKHR VkAabbPositionsNV;

- minX is the x position of one opposing corner of a bounding box.
- minY is the y position of one opposing corner of a bounding box.
- minZ is the z position of one opposing corner of a bounding box.
- maxX is the x position of the other opposing corner of a bounding box.
- maxY is the y position of the other opposing corner of a bounding box.
- maxZ is the z position of the other opposing corner of a bounding box.

Valid Usage

- VUID-VkAabbPositionsKHR-minX-03546
  minX must be less than or equal to maxX

- VUID-VkAabbPositionsKHR-minY-03547
  minY must be less than or equal to maxY

- VUID-VkAabbPositionsKHR-minZ-03548
  minZ must be less than or equal to maxZ

The VkAccelerationStructureGeometryInstancesDataKHR structure is defined as:

typedef struct VkAccelerationStructureGeometryInstancesDataKHR {
    VkStructureType sType;
    const void* pNext;
    VkBool32 arrayOfPointers;
    VkDeviceOrHostAddressConstKHR data;
} VkAccelerationStructureGeometryInstancesDataKHR;

- sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.

• arrayOfPointers specifies whether data is used as an array of addresses or just an array.

• data is either the address of an array of device or host addresses referencing individual VkAccelerationStructureInstanceKHR structures if arrayOfPointers is VK_TRUE, or the address of an array of VkAccelerationStructureInstanceKHR structures.

### Valid Usage (Implicit)

- VUID-VkAccelerationStructureGeometryInstancesDataKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR

- VUID-VkAccelerationStructureGeometryInstancesDataKHR-pNext-pNext
  pNext must be NULL

**Acceleration structure instances** can be built into top-level acceleration structures. Each acceleration structure instance is a separate entry in the top-level acceleration structure which includes all the geometry of a bottom-level acceleration structure at a transformed location. Multiple instances can point to the same bottom level acceleration structure.

An acceleration structure instance is defined by the structure:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureInstanceKHR {
    VkTransformMatrixKHR transform;
    uint32_t instanceCustomIndex:24;
    uint32_t mask:8;
    uint32_t instanceShaderBindingTableRecordOffset:24;
    VkGeometryInstanceFlagsKHR flags:8;
    uint64_t accelerationStructureReference;
} VkAccelerationStructureInstanceKHR;
```

or the equivalent

```c
// Provided by VK_NV_ray_tracing
typedef VkAccelerationStructureInstanceKHR VkAccelerationStructureInstanceNV;
```

- transform is a VkTransformMatrixKHR structure describing a transformation to be applied to the acceleration structure.

- instanceCustomIndex is a 24-bit user-specified index value accessible to ray shaders in the InstanceCustomIndexKHR built-in.

- mask is an 8-bit visibility mask for the geometry. The instance may only be hit if rayMask & instance.mask != 0

- instanceShaderBindingTableRecordOffset is a 24-bit offset used in calculating the hit shader binding table index.
• **flags** is an 8-bit mask of **VkGeometryInstanceFlagBitsKHR** values to apply to this instance.

• **accelerationStructureReference** is either:
  ◦ a device address containing the value obtained from
    **vkGetAccelerationStructureDeviceAddressKHR** or **vkGetAccelerationStructureHandleNV**
    (used by device operations which reference acceleration structures) or,
  ◦ a **VkAccelerationStructureKHR** object (used by host operations which reference acceleration structures).

The C language spec does not define the ordering of bit-fields, but in practice, this struct produces the correct layout with existing compilers. The intended bit pattern is for the following:

• **instanceCustomIndex** and **mask** occupy the same memory as if a single **int32_t** was specified in their place
  ◦ **instanceCustomIndex** occupies the 24 least significant bits of that memory
  ◦ **mask** occupies the 8 most significant bits of that memory

• **instanceShaderBindingTableRecordOffset** and **flags** occupy the same memory as if a single **int32_t** was specified in their place
  ◦ **instanceShaderBindingTableRecordOffset** occupies the 24 least significant bits of that memory
  ◦ **flags** occupies the 8 most significant bits of that memory

If a compiler produces code that diverges from that pattern, applications **must** employ another method to set values according to the correct bit pattern.

### Valid Usage (Implicit)

• **VUID-VkAccelerationStructureInstanceKHR-flags-parameter**
  **flags** **must** be a valid combination of **VkGeometryInstanceFlagBitsKHR** values

Possible values of **flags** in the instance modifying the behavior of that instance are:
typedef enum VkGeometryInstanceFlagBitsKHR {
    VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR = 0x00000001,
    VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_KHR = 0x00000002,
    VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_KHR = 0x00000004,
    VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_KHR = 0x00000008,

    // Provided by VK_NV_ray_tracing
    VK_GEOMETRY_INSTANCE_TRIANGLE_CULL_DISABLE_BIT_NV =
    VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_NV =
    VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_NV =
    VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_KHR,
    // Provided by VK_NV_ray_tracing
    VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_NV =
    VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_KHR,
} VkGeometryInstanceFlagBitsKHR;

or the equivalent

// Provided by VK_NV_ray_tracing
typedef VkGeometryInstanceFlagBitsKHR VkGeometryInstanceFlagBitsNV;

• VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR disables face culling for this instance.

• VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_KHR indicates that the front face of the triangle for culling purposes is the face that is counter clockwise in object space relative to the ray origin. Because the facing is determined in object space, an instance transform matrix does not change the winding, but a geometry transform does.

• VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_KHR causes this instance to act as though VK_GEOMETRY_OPAQUE_BIT_KHR were specified on all geometries referenced by this instance. This behavior can be overridden by the SPIR-V NoOpaqueKHR ray flag.

• VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_KHR causes this instance to act as though VK_GEOMETRY_OPAQUE_BIT_KHR were not specified on all geometries referenced by this instance. This behavior can be overridden by the SPIR-V OpaqueKHR ray flag.

VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_KHR and VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_KHR must not be used in the same flag.

Chapter 36. Acceleration Structures | 1917
typedef VkGeometryInstanceFlagsKHR VkGeometryInstanceFlagsNV;

VkGeometryInstanceFlagsKHR is a bitmask type for setting a mask of zero or more VkGeometryInstanceFlagBitsKHR.

VkAccelerationStructureBuildRangeInfoKHR is defined as:

typedef struct VkAccelerationStructureBuildRangeInfoKHR {
    uint32_t    primitiveCount;
    uint32_t    primitiveOffset;
    uint32_t    firstVertex;
    uint32_t    transformOffset;
} VkAccelerationStructureBuildRangeInfoKHR;

• primitiveCount defines the number of primitives for a corresponding acceleration structure geometry.
• primitiveOffset defines an offset in bytes into the memory where primitive data is defined.
• firstVertex is the index of the first vertex to build from for triangle geometry.
• transformOffset defines an offset in bytes into the memory where a transform matrix is defined.

The primitive count and primitive offset are interpreted differently depending on the VkGeometryTypeKHR used:

• For geometries of type VK_GEOMETRY_TYPE_TRIANGLES_KHR, primitiveCount is the number of triangles to be built, where each triangle is treated as 3 vertices.
  ◦ If the geometry uses indices, primitiveCount × 3 indices are consumed from VkAccelerationStructureGeometryTrianglesDataKHR::indexData, starting at an offset of primitiveOffset. The value of firstVertex is added to the index values before fetching vertices.
  ◦ If the geometry does not use indices, primitiveCount × 3 vertices are consumed from VkAccelerationStructureGeometryTrianglesDataKHR::vertexData, starting at an offset of primitiveOffset + VkAccelerationStructureGeometryTrianglesDataKHR::vertexStride × firstVertex.
  ◦ A single VkTransformMatrixKHR structure is consumed from VkAccelerationStructureGeometryTrianglesDataKHR::transformData, at an offset of transformOffset. This transformation matrix is used by all triangles.

• For geometries of type VK_GEOMETRY_TYPE_AABBS_KHR, primitiveCount is the number of axis-aligned bounding boxes. primitiveCount VkAabbPositionsKHR structures are consumed from VkAccelerationStructureGeometryAabbsDataKHR::data, starting at an offset of primitiveOffset.

• For geometries of type VK_GEOMETRY_TYPE_INSTANCES_KHR, primitiveCount is the number of acceleration structures. primitiveCount VkAccelerationStructureInstanceKHR structures are consumed from VkAccelerationStructureGeometryInstancesDataKHR::data, starting at an offset...
of `primitiveOffset`.

## Valid Usage

- **VUID-VkAccelerationStructureBuildRangeInfoKHR-primitiveOffset-03656**
  For geometries of type `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, if the geometry uses indices, the offset `primitiveOffset` from `VkAccelerationStructureGeometryTrianglesDataKHR::indexData` must be a multiple of the element size of `VkAccelerationStructureGeometryTrianglesDataKHR::indexType`

- **VUID-VkAccelerationStructureBuildRangeInfoKHR-primitiveOffset-03657**
  For geometries of type `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, if the geometry doesn’t use indices, the offset `primitiveOffset` from `VkAccelerationStructureGeometryTrianglesDataKHR::vertexData` must be a multiple of the component size of `VkAccelerationStructureGeometryTrianglesDataKHR::vertexFormat`

- **VUID-VkAccelerationStructureBuildRangeInfoKHR-transformOffset-03658**
  For geometries of type `VK_GEOMETRY_TYPE_TRIANGLES_KHR`, the offset `transformOffset` from `VkAccelerationStructureGeometryTrianglesDataKHR::transformData` must be a multiple of 16

- **VUID-VkAccelerationStructureBuildRangeInfoKHR-primitiveOffset-03659**
  For geometries of type `VK_GEOMETRY_TYPE_AABBS_KHR`, the offset `primitiveOffset` from `VkAccelerationStructureGeometryAabbsDataKHR::data` must be a multiple of 8

- **VUID-VkAccelerationStructureBuildRangeInfoKHR-primitiveOffset-03660**
  For geometries of type `VK_GEOMETRY_TYPE_INSTANCES_KHR`, the offset `primitiveOffset` from `VkAccelerationStructureGeometryInstancesDataKHR::data` must be a multiple of 16

### 36.1.7. Copying Acceleration Structures

An additional command exists for copying acceleration structures without updating their contents. The acceleration structure object can be compacted in order to improve performance. Before copying, an application must query the size of the resulting acceleration structure.

To query acceleration structure size parameters call:

```c
// Provided by VK_KHR_acceleration_structure
void vkCmdWriteAccelerationStructuresPropertiesKHR(
    VkCommandBuffer       commandBuffer, 
    uint32_t              accelerationStructureCount, 
    const VkAccelerationStructureKHR* pAccelerationStructures, 
    VkQueryType           queryType, 
    VkQueryPool           queryPool, 
    uint32_t              firstQuery);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `accelerationStructureCount` is the count of acceleration structures for which to query the property.
• \texttt{pAccelerationStructures} is a pointer to an array of existing previously built acceleration structures.

• \texttt{queryType} is a \texttt{VkQueryType} value specifying the type of queries managed by the pool.

• \texttt{queryPool} is the query pool that will manage the results of the query.

• \texttt{firstQuery} is the first query index within the query pool that will contain the \texttt{accelerationStructureCount} number of results.

Accesses to any of the acceleration structures listed in \texttt{pAccelerationStructures} must be synchronized with the \texttt{VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR} pipeline stage and an access type of \texttt{VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR}.

### Valid Usage

- \texttt{VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-queryPool-02493}
  \texttt{queryPool} must have been created with a \texttt{queryType} matching \texttt{queryType}

- \texttt{VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-queryPool-02494}
  The queries identified by \texttt{queryPool} and \texttt{firstQuery} must be unavailable

- \texttt{VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-buffer-03736}
  The buffer used to create each acceleration structure in \texttt{pAccelerationStructures} must be bound to device memory

- \texttt{VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-query-04880}
  The sum of \texttt{query} plus \texttt{accelerationStructureCount} must be less than or equal to the number of queries in \texttt{queryPool}

- \texttt{VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-accelerationStructures-03431}
  All acceleration structures in \texttt{pAccelerationStructures} must have been built with \texttt{VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR} if \texttt{queryType} is \texttt{VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR}

- \texttt{VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-queryType-03432}
  \texttt{queryType} must be \texttt{VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR} or \texttt{VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR}
Valid Usage (Implicit)

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-pAccelerationStructures-parameter
  
  `pAccelerationStructures` must be a valid pointer to an array of `accelerationStructureCount` valid `VkAccelerationStructureKHR` handles

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-queryType-parameter
  
  `queryType` must be a valid `VkQueryType` value

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-queryPool-parameter
  
  `queryPool` must be a valid `VkQueryPool` handle

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-renderpass
  
  This command must only be called outside of a render pass instance

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-accelerationStructureCount-arraylength
  
  `accelerationStructureCount` must be greater than 0

- VUID-vkCmdWriteAccelerationStructuresPropertiesKHR-commonparent
  
  Each of `commandBuffer`, `queryPool`, and the elements of `pAccelerationStructures` must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To query acceleration structure size parameters call:
// Provided by VK_NV_ray_tracing

```c
void vkCmdWriteAccelerationStructuresPropertiesNV(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    accelerationStructureCount,
    const VkAccelerationStructureNV*            pAccelerationStructures,
    VkQueryType                                 queryType,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `accelerationStructureCount` is the count of acceleration structures for which to query the property.
- `pAccelerationStructures` is a pointer to an array of existing previously built acceleration structures.
- `queryType` is a `VkQueryType` value specifying the type of queries managed by the pool.
- `queryPool` is the query pool that will manage the results of the query.
- `firstQuery` is the first query index within the query pool that will contain the `accelerationStructureCount` number of results.

Accesses to any of the acceleration structures listed in `pAccelerationStructures` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR`.

### Valid Usage

- VUID-vkCmdWriteAccelerationStructuresPropertiesNV-queryPool-03755
  * queryPool must have been created with a `queryType` matching `queryType`

- VUID-vkCmdWriteAccelerationStructuresPropertiesNV-queryPool-03756
  * The queries identified by `queryPool` and `firstQuery` must be unavailable

- VUID-vkCmdWriteAccelerationStructuresPropertiesNV-accelerationStructure-03757
  * accelerationStructure must be bound completely and contiguously to a single `VkDeviceMemory` object via `vkBindAccelerationStructureMemoryNV`

- VUID-vkCmdWriteAccelerationStructuresPropertiesNV-accelerationStructures-03431
  * All acceleration structures in `pAccelerationStructures` must have been built with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR` if `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV`

- VUID-vkCmdWriteAccelerationStructuresPropertiesNV-queryType-03432
  * `queryType` must be `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_NV`
Valid Usage (Implicit)

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-pAccelerationStructures-parameter**
  
  `pAccelerationStructures` must be a valid pointer to an array of `accelerationStructureCount` valid `VkAccelerationStructureNV` handles

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-queryType-parameter**
  
  `queryType` must be a valid `VkQueryType` value

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-queryPool-parameter**
  
  `queryPool` must be a valid `VkQueryPool` handle

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-renderpass**
  
  This command must only be called outside of a render pass instance

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-accelerationStructureCount-arraylength**
  
  `accelerationStructureCount` must be greater than 0

- **VUID-vkCmdWriteAccelerationStructuresPropertiesNV-commonparent**
  
  Each of `commandBuffer`, `queryPool`, and the elements of `pAccelerationStructures` must have been created, allocated, or retrieved from the same `VkDevice`

---

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

---

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To copy an acceleration structure call:
// Provided by VK_NV_ray_tracing
void vkCmdCopyAccelerationStructureNV(
    VkCommandBuffer                          commandBuffer,
    VkAccelerationStructureNV               dst,
    VkAccelerationStructureNV               src,
    VkCopyAccelerationStructureModeKHR      mode);

- `commandBuffer` is the command buffer into which the command will be recorded.
- `dst` is the target acceleration structure for the copy.
- `src` is the source acceleration structure for the copy.
- `mode` is a `VkCopyAccelerationStructureModeKHR` value specifying additional operations to perform during the copy.

Accesses to `src` and `dst` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR` or `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR` as appropriate.

## Valid Usage

- VUID-vkCmdCopyAccelerationStructureNV-mode-03410
  mode must be `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR` or `VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_KHR`

- VUID-vkCmdCopyAccelerationStructureNV-src-03411
  If `mode` is `VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR`, `src` must have been built with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR`

- VUID-vkCmdCopyAccelerationStructureNV-buffer-03718
  The buffer used to create `src` must be bound to device memory

- VUID-vkCmdCopyAccelerationStructureNV-buffer-03719
  The buffer used to create `dst` must be bound to device memory
Valid Usage (Implicit)

- **VUID-vkCmdCopyAccelerationStructureNV-commandBuffer-parameter**
  
  *commandBuffer must be a valid* VkCommandBuffer *handle*

- **VUID-vkCmdCopyAccelerationStructureNV-dst-parameter**
  
  *dst must be a valid* VkAccelerationStructureNV *handle*

- **VUID-vkCmdCopyAccelerationStructureNV-src-parameter**
  
  *src must be a valid* VkAccelerationStructureNV *handle*

- **VUID-vkCmdCopyAccelerationStructureNV-mode-parameter**
  
  *mode must be a valid* VkCopyAccelerationStructureModeKHR *value*

- **VUID-vkCmdCopyAccelerationStructureNV-commandBuffer-recording**
  
  *commandBuffer must be in the recording state*

- **VUID-vkCmdCopyAccelerationStructureNV-commandBuffer-cmdpool**
  
  The *VkCommandPool* that *commandBuffer* was allocated from *must* support compute operations

- **VUID-vkCmdCopyAccelerationStructureNV-renderpass**
  
  This command *must* only be called outside of a render pass instance

- **VUID-vkCmdCopyAccelerationStructureNV-commonparent**
  
  Each of *commandBuffer*, *dst*, and *src* *must* have been created, allocated, or retrieved from the same *VkDevice*

Host Synchronization

- Host access to *commandBuffer* *must* be externally synchronized

- Host access to the *VkCommandPool* that *commandBuffer* was allocated from *must* be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To copy an acceleration structure call:

```c
// Provided by VK_KHR_acceleration_structure
void vkCmdCopyAccelerationStructureKHR(
    VkCommandBuffer commandBuffer,
    const VkCopyAccelerationStructureInfoKHR* pInfo);
```
• `commandBuffer` is the command buffer into which the command will be recorded.

• `pInfo` is a pointer to a `VkCopyAccelerationStructureInfoKHR` structure defining the copy operation.

Accesses to `pInfo->src` and `pInfo->dst` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR` or `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR` as appropriate.

---

**Valid Usage**

- VUID-vkCmdCopyAccelerationStructureKHR-buffer-03737
  The buffer used to create `pInfo->src` must be bound to device memory

- VUID-vkCmdCopyAccelerationStructureKHR-buffer-03738
  The buffer used to create `pInfo->dst` must be bound to device memory

---

**Valid Usage (Implicit)**

- VUID-vkCmdCopyAccelerationStructureKHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdCopyAccelerationStructureKHR-pInfo-parameter
  `pInfo` must be a valid pointer to a valid `VkCopyAccelerationStructureInfoKHR` structure

- VUID-vkCmdCopyAccelerationStructureKHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdCopyAccelerationStructureKHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations

- VUID-vkCmdCopyAccelerationStructureKHR-renderpass
  This command must only be called outside of a render pass instance

---

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized
The `VkCopyAccelerationStructureInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkCopyAccelerationStructureInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureKHR src;
    VkAccelerationStructureKHR dst;
    VkCopyAccelerationStructureModeKHR mode;
} VkCopyAccelerationStructureInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `src` is the source acceleration structure for the copy.
- `dst` is the target acceleration structure for the copy.
- `mode` is a `VkCopyAccelerationStructureModeKHR` value that specifies additional operations to perform during the copy.

**Valid Usage**

- VUID-VkCopyAccelerationStructureInfoKHR-mode-03410
  ```
  mode must be VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR or
  VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_KHR
  ```

- VUID-VkCopyAccelerationStructureInfoKHR-src-03411
  ```
  If mode is VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR, src must have been built with
  VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR
  ```

- VUID-VkCopyAccelerationStructureInfoKHR-buffer-03718
  ```
  The buffer used to create src must be bound to device memory
  ```

- VUID-VkCopyAccelerationStructureInfoKHR-buffer-03719
  ```
  The buffer used to create dst must be bound to device memory
  ```
Valid Usage (Implicit)

- **VUID-VkCopyAccelerationStructureInfoKHR-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_INFO_KHR`
- **VUID-VkCopyAccelerationStructureInfoKHR-pNext-pNext**
  - `pNext` must be `NULL`
- **VUID-VkCopyAccelerationStructureInfoKHR-src-parameter**
  - `src` must be a valid `VkAccelerationStructureKHR` handle
- **VUID-VkCopyAccelerationStructureInfoKHR-dst-parameter**
  - `dst` must be a valid `VkAccelerationStructureKHR` handle
- **VUID-VkCopyAccelerationStructureInfoKHR-mode-parameter**
  - `mode` must be a valid `VkCopyAccelerationStructureModeKHR` value
- **VUID-VkCopyAccelerationStructureInfoKHR-commonparent**
  - Both of `dst`, and `src` must have been created, allocated, or retrieved from the same `VkDevice`

Possible values of `mode` specifying additional operations to perform during the copy, are:

```c
// Provided by VK_KHR_acceleration_structure
typedef enum VkCopyAccelerationStructureModeKHR {
    VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_KHR = 0,
    VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR = 1,
    VK_COPY_ACCELERATION_STRUCTURE_MODE_SERIALIZE_KHR = 2,
    VK_COPY_ACCELERATION_STRUCTURE_MODE_DESERIALIZE_KHR = 3,
} VkCopyAccelerationStructureModeKHR;
```

or the equivalent

```c
// Provided by VK_NV_ray_tracing
typedef VkCopyAccelerationStructureModeKHR VkCopyAccelerationStructureModeNV;
```

- **`VK_COPY_ACCELERATION_STRUCTURE_MODE_CLONE_KHR`** creates a direct copy of the acceleration structure specified in `src` into the one specified by `dst`. The `dst` acceleration structure must have been created with the same parameters as `src`.
- **`VK_COPY_ACCELERATION_STRUCTURE_MODE_COMPACT_KHR`** creates a more compact version of an acceleration structure `src` into `dst`. The acceleration structure `dst` must have been created with a size at least as large as that returned by `vkCmdWriteAccelerationStructuresPropertiesKHR` or
vkWriteAccelerationStructuresPropertiesKHR after the build of the acceleration structure specified by src.

- **VK_COPY_ACCELERATION_STRUCTURE_MODE_SERIALIZE_KHR** serializes the acceleration structure to a semi-opaque format which can be reloaded on a compatible implementation.
- **VK_COPY_ACCELERATION_STRUCTURE_MODE_DESERIALIZE_KHR** deserializes the semi-opaque serialization format in the buffer to the acceleration structure.

To copy an acceleration structure to device memory call:

```c
// Provided by VK_KHR_acceleration_structure
void vkCmdCopyAccelerationStructureToMemoryKHR(
    VkCommandBuffer                             commandBuffer,
    const VkCopyAccelerationStructureToMemoryInfoKHR* pInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pInfo` is an a pointer to a `VkCopyAccelerationStructureToMemoryInfoKHR` structure defining the copy operation.

Accesses to `pInfo->src` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR`. Accesses to the buffer indicated by `pInfo->dst.deviceAddress` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_TRANSFER_WRITE_BIT`.

This command produces the same results as `vkCopyAccelerationStructureToMemoryKHR`, but writes its result to a device address, and is executed on the device rather than the host. The output may not necessarily be bit-for-bit identical, but it can be equally used by either `vkCmdCopyMemoryToAccelerationStructureKHR` or `vkCopyMemoryToAccelerationStructureKHR`.

The defined header structure for the serialized data consists of:

- **VK_UUID_SIZE** bytes of data matching `VkPhysicalDeviceIDProperties::driverUUID`
- **VK_UUID_SIZE** bytes of data identifying the compatibility for comparison using `vkGetDeviceAccelerationStructureCompatibilityKHR`
- A 64-bit integer of the total size matching the value queried using `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`
- A 64-bit integer of the deserialized size to be passed in to `VkAccelerationStructureCreateInfoKHR::size`
- A 64-bit integer of the count of the number of acceleration structure handles following. This will be zero for a bottom-level acceleration structure. For top-level acceleration structures this number is implementation-dependent; the number of and ordering of the handles may not match the instance descriptions which were used to build the acceleration structure.

The corresponding handles matching the values returned by `vkGetAccelerationStructureDeviceAddressKHR` or `vkGetAccelerationStructureHandleNV` are tightly
packed in the buffer following the count. The application is expected to store a mapping between those handles and the original application-generated bottom-level acceleration structures to provide when deserializing. The serialized data is written to the buffer (or read from the buffer) according to the host endianness.

### Valid Usage

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-pInfo-03739
  pInfo->dst.deviceAddress must be a valid device address for a buffer bound to device memory.

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-pInfo-03740
  pInfo->dst.deviceAddress must be aligned to 256 bytes

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-pInfo-03741
  If the buffer pointed to by pInfo->dst.deviceAddress is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-None-03559
  The buffer used to create pInfo->src must be bound to device memory

### Valid Usage (Implicit)

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-pInfo-parameter
  pInfo must be a valid pointer to a valid VkCopyAccelerationStructureToMemoryInfoKHR structure

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support compute operations

- VUID-vkCmdCopyAccelerationStructureToMemoryKHR-renderpass
  This command must only be called outside of a render pass instance

### Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

// Provided by VK_KHR_acceleration_structure

typedef struct VkCopyAccelerationStructureToMemoryInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkAccelerationStructureKHR src;
    VkDeviceOrHostAddressKHR dst;
    VkCopyAccelerationStructureModeKHR mode;
} VkCopyAccelerationStructureToMemoryInfoKHR;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **src** is the source acceleration structure for the copy
- **dst** is the device or host address to memory which is the target for the copy
- **mode** is a VkCopyAccelerationStructureModeKHR value that specifies additional operations to perform during the copy.

Valid Usage

- VUID-VkCopyAccelerationStructureToMemoryInfoKHR-dst-03561
  The memory pointed to by **dst** must be at least as large as the serialization size of **src**, as reported by `vkWriteAccelerationStructuresPropertiesKHR` or `vkCmdWriteAccelerationStructuresPropertiesKHR` with a query type of `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`

- VUID-VkCopyAccelerationStructureToMemoryInfoKHR-mode-03412
  **mode** must be `VK_COPY_ACCELERATION_STRUCTURE_MODE_SERIALIZE_KHR`
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_TO_MEMORY_INFO_KHR`
- `pNext` must be `NULL`
- `src` must be a valid `VkAccelerationStructureKHR` handle
- `mode` must be a valid `VkCopyAccelerationStructureModeKHR` value

To copy device memory to an acceleration structure call:

```c
// Provided by VK_KHR_acceleration_structure
void vkCmdCopyMemoryToAccelerationStructureKHR(
    VkCommandBuffer                             commandBuffer,
    const VkCopyMemoryToAccelerationStructureInfoKHR* pInfo);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pInfo` is a pointer to a `VkCopyMemoryToAccelerationStructureInfoKHR` structure defining the copy operation.

Accesses to `pInfo->dst` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR`. Accesses to the buffer indicated by `pInfo->src.deviceAddress` must be synchronized with the `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` pipeline stage and an access type of `VK_ACCESS_TRANSFER_READ_BIT`.

This command can accept acceleration structures produced by either `vkCmdCopyAccelerationStructureToMemoryKHR` or `vkCopyAccelerationStructureToMemoryKHR`.

The structure provided as input to deserialize is as described in `vkCmdCopyAccelerationStructureToMemoryKHR`, with any acceleration structure handles filled in with the newly-queried handles to bottom level acceleration structures created before deserialization. These do not need to be built at deserialize time, but must be created.
Valid Usage

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-pInfo-03742
  pInfo-&gt;src.deviceAddress must be a valid device address for a buffer bound to device memory.

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-pInfo-03743
  pInfo-&gt;src.deviceAddress must be aligned to 256 bytes

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-pInfo-03744
  If the buffer pointed to by pInfo-&gt;src.deviceAddress is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-buffer-03745
  The buffer used to create pInfo-&gt;dst must be bound to device memory

Valid Usage (Implicit)

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-pInfo-parameter
  pInfo must be a valid pointer to a valid VkCopyMemoryToAccelerationStructureInfoKHR structure

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support compute operations

- VUID-vkCmdCopyMemoryToAccelerationStructureKHR-renderpass
  This command must only be called outside of a render pass instance

Host Synchronization

- Host access to commandBuffer must be externally synchronized

- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized
Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkCopyMemoryToAccelerationStructureInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkCopyMemoryToAccelerationStructureInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkDeviceOrHostAddressConstKHR src;
    VkAccelerationStructureKHR dst;
    VkCopyAccelerationStructureModeKHR mode;
} VkCopyMemoryToAccelerationStructureInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `src` is the device or host address to memory containing the source data for the copy.
- `dst` is the target acceleration structure for the copy.
- `mode` is a `VkCopyAccelerationStructureModeKHR` value that specifies additional operations to perform during the copy.

Valid Usage

- VUID-VkCopyMemoryToAccelerationStructureInfoKHR-mode-03413
  `mode` must be `VK_COPY_ACCELERATION_STRUCTURE_MODE_DESERIALIZE_KHR`

- VUID-VkCopyMemoryToAccelerationStructureInfoKHR-pInfo-03414
  The data in `src` must have a format compatible with the destination physical device as returned by `vkGetDeviceAccelerationStructureCompatibilityKHR`

- VUID-VkCopyMemoryToAccelerationStructureInfoKHR-dst-03746
  `dst` must have been created with a size greater than or equal to that used to serialize the data in `src`
Valid Usage (Implicit)

- **VUID-VkCopyMemoryToAccelerationStructureInfoKHR-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_COPY_MEMORY_TO_ACCELERATION_STRUCTURE_INFO_KHR*

- **VUID-VkCopyMemoryToAccelerationStructureInfoKHR-pNext-pNext**

  *pNext must be NULL*

- **VUID-VkCopyMemoryToAccelerationStructureInfoKHR-dst-parameter**

  *dst must be a valid VkAccelerationStructureKHR handle*

- **VUID-VkCopyMemoryToAccelerationStructureInfoKHR-mode-parameter**

  *mode must be a valid VkCopyAccelerationStructureModeKHR value*

To check if a serialized acceleration structure is compatible with the current device call:

```c
// Provided by VK_KHR_acceleration_structure
void vkGetDeviceAccelerationStructureCompatibilityKHR(
    VkDevice device,
    const VkAccelerationStructureVersionInfoKHR* pVersionInfo,
    VkAccelerationStructureCompatibilityKHR* pCompatibility);
```

- **device** is the device to check the version against.

- **pVersionInfo** points to the VkAccelerationStructureVersionInfoKHR version information to check against the device.

- **pCompatibility** points to the VkAccelerationStructureCompatibilityKHR compatibility return value.

Valid Usage

- **VUID-vkGetDeviceAccelerationStructureCompatibilityKHR-rayTracingPipeline-03661**

  The rayTracingPipeline or rayQuery feature must be enabled

Valid Usage (Implicit)

- **VUID-vkGetDeviceAccelerationStructureCompatibilityKHR-device-parameter**

  *device must be a valid VkDevice handle*

- **VUID-vkGetDeviceAccelerationStructureCompatibilityKHR-pVersionInfo-parameter**

  *pVersionInfo must be a valid pointer to a valid VkAccelerationStructureVersionInfoKHR structure*

- **VUID-vkGetDeviceAccelerationStructureCompatibilityKHR-pCompatibility-parameter**

  *pCompatibility must be a valid pointer to a VkAccelerationStructureCompatibilityKHR value*
The VkAccelerationStructureVersionInfoKHR structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkAccelerationStructureVersionInfoKHR {
    VkStructureType    sType;
    const void*        pNext;
    const uint8_t*     pVersionData;
} VkAccelerationStructureVersionInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `pVersionData` is a pointer to the version header of an acceleration structure as defined in `vkCmdCopyAccelerationStructureToMemoryKHR`.

**Note**

`pVersionData` is a *pointer* to an array of 2*VK_UUID_SIZE uint8_t values instead of two `VK_UUID_SIZE` arrays as the expected use case for this member is to be pointed at the header of an previously serialized acceleration structure (via `vkCmdCopyAccelerationStructureToMemoryKHR` or `vkCopyAccelerationStructureToMemoryKHR`) that is loaded in memory. Using arrays would necessitate extra memory copies of the UUIDs.

### Valid Usage (Implicit)

- **VUID-VkAccelerationStructureVersionInfoKHR-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_VERSION_INFO_KHR`

- **VUID-VkAccelerationStructureVersionInfoKHR-pNext-pNext**
  
  `pNext` must be NULL

- **VUID-VkAccelerationStructureVersionInfoKHR-pVersionData-parameter**
  
  `pVersionData` must be a valid pointer to an array of 2*VK_UUID_SIZE uint8_t values

Possible values of `pCompatibility` returned by `vkGetDeviceAccelerationStructureCompatibilityKHR` are:

```c
// Provided by VK_KHR_acceleration_structure
typedef enum VkAccelerationStructureCompatibilityKHR {
    VK_ACCELERATION_STRUCTURE_COMPATIBILITY_COMPATIBLE_KHR = 0,
    VK_ACCELERATION_STRUCTURE_COMPATIBILITY_INCOMPATIBLE_KHR = 1,
} VkAccelerationStructureCompatibilityKHR;
```

- **VK_ACCELERATION_STRUCTURE_COMPATIBILITY_COMPATIBLE_KHR** when the `pVersion` version acceleration structure is compatible with `device`.
- **VK_ACCELERATION_STRUCTURE_COMPATIBILITY_INCOMPATIBLE_KHR** when the `pVersion` version
acceleration structure is not compatible with device.

## 36.2. Host Acceleration Structure Operations

Implementations are also required to provide host implementations of the acceleration structure operations if the `accelerationStructureHostCommands` feature is enabled:

- `vkBuildAccelerationStructuresKHR` corresponding to `vkCmdBuildAccelerationStructuresKHR`
- `vkCopyAccelerationStructureKHR` corresponding to `vkCmdCopyAccelerationStructureKHR`
- `vkCopyAccelerationStructureToMemoryKHR` corresponding to `vkCmdCopyAccelerationStructureToMemoryKHR`
- `vkCopyMemoryToAccelerationStructureKHR` corresponding to `vkCmdCopyMemoryToAccelerationStructureKHR`
- `vkWriteAccelerationStructuresPropertiesKHR` corresponding to `vkCmdWriteAccelerationStructuresPropertiesKHR`

These commands are functionally equivalent to their device counterparts, except that they are executed on the host timeline, rather than being enqueued into command buffers.

All acceleration structures used by the host commands **must** be bound to host-visible memory, and all input data for acceleration structure builds **must** be referenced using host addresses instead of device addresses. Applications are not required to map acceleration structure memory when using the host commands.

**Note**
The `vkBuildAccelerationStructuresKHR` and `vkCmdBuildAccelerationStructuresKHR` may use different algorithms, and thus are not required to produce identical structures. The structures produced by these two commands **may** exhibit different memory footprints or traversal performance, but should strive to be similar where possible.

Apart from these details, the host and device operations are interchangeable. For example, an application **can** use `vkBuildAccelerationStructuresKHR` to build a structure, compact it on the device using `vkCmdCopyAccelerationStructureKHR`, and serialize the result using `vkCopyAccelerationStructureToMemoryKHR`.

To build acceleration structures on the host, call:

```c
// Provided by VK_KHR_acceleration_structure
VkResult vkBuildAccelerationStructuresKHR(
  VkDevice device,        // Provided
  VkDeferredOperationKHR deferredOperation, // Provided
  uint32_t infoCount,     // Provided
  const VkAccelerationStructureBuildGeometryInfoKHR* pInfos,
  const VkAccelerationStructureBuildRangeInfoKHR* ppBuildRangeInfos);
```
device is the VkDevice for which the acceleration structures are being built.

deferedOperation is an optional VkDeferredOperationKHR to request deferral for this command.

infoCount is the number of acceleration structures to build. It specifies the number of the pInfo structures and ppBuildRangeInfos pointers that must be provided.

pInfos is a pointer to an array of infoCount VkAccelerationStructureBuildGeometryInfoKHR structures defining the geometry used to build each acceleration structure.

ppBuildRangeInfos is an array of infoCount pointers to arrays of VkAccelerationStructureBuildRangeInfoKHR structures. Each ppBuildRangeInfos[i] is an array of pInfo[i].geometryCount VkAccelerationStructureBuildRangeInfoKHR structures defining dynamic offsets to the addresses where geometry data is stored, as defined by pInfo[i].

This command fulfills the same task as vkCmdBuildAccelerationStructuresKHR but is executed by the host.

The vkBuildAccelerationStructuresKHR command provides the ability to initiate multiple acceleration structures builds, however there is no ordering or synchronization implied between any of the individual acceleration structure builds.

Note

This means that an application cannot build a top-level acceleration structure in the same vkBuildAccelerationStructuresKHR call as the associated bottom-level or instance acceleration structures are being built. There also cannot be any memory aliasing between any acceleration structure memories or scratch memories being used by any of the builds.
Valid Usage

- **VUID-vkBuildAccelerationStructuresKHR-mode-04628**
  The `mode` member of each element of `pInfos` **must** be a valid `VkBuildAccelerationStructureModeKHR` value.

- **VUID-vkBuildAccelerationStructuresKHR-srcAccelerationStructure-04629**
  If the `srcAccelerationStructure` member of any element of `pInfos` is not `VK_NULL_HANDLE`, the `srcAccelerationStructure` member **must** be a valid `VkAccelerationStructureKHR` handle.

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-04630**
  For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member **must not be** `VK_NULL_HANDLE`.

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-03403**
  The `srcAccelerationStructure` member of any element of `pInfos` **must not be** the same acceleration structure as the `dstAccelerationStructure` member of any other element of `pInfos`.

- **VUID-vkBuildAccelerationStructuresKHR-dstAccelerationStructure-03698**
  The `dstAccelerationStructure` member of any element of `pInfos` **must not be** the same acceleration structure as the `dstAccelerationStructure` member of any other element of `pInfos`.

- **VUID-vkBuildAccelerationStructuresKHR-dstAccelerationStructure-03800**
  The `dstAccelerationStructure` member of any element of `pInfos` **must** be a valid `VkAccelerationStructureKHR` handle.

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-03699**
  For each element of `pInfos`, if its `type` member is `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR`, its `dstAccelerationStructure` member **must** have been created with a value of `VkAccelerationStructureCreateInfoKHR::type` equal to either `VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR` or `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR`.

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-03700**
  For each element of `pInfos`, if its `type` member is `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR`, its `dstAccelerationStructure` member **must** have been created with a value of `VkAccelerationStructureCreateInfoKHR::type` equal to either `VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR` or `VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR`.

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-03663**
  For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, **inactive** primitives in its `srcAccelerationStructure` member **must not be** made active.

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-03664**
  For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, **active** primitives in its `srcAccelerationStructure` member **must not be** made inactive.
The `dstAccelerationStructure` member of any element of `pInfos` **must** not be referenced by the `geometry.instances.data` member of any element of `pGeometries` or `ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR` in any other element of `pInfos`.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command **must** not overlap the memory backing the `srcAccelerationStructure` member of any other element of `pInfos` with a `mode` equal to `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, which is accessed by this command.

The range of memory backing the `dstAccelerationStructure` member of any element of `pInfos` that is accessed by this command **must** not overlap the memory backing the `scratchData` member of any element of `pInfos` (including the same element), which is accessed by this command.

The range of memory backing the `scratchData` member of any element of `pInfos` that is accessed by this command **must** not overlap the memory backing the `scratchData` member of any other element of `pInfos`, which is accessed by this command.

The range of memory backing the `scratchData` member of any element of `pInfos` that is accessed by this command **must** not overlap the memory backing any acceleration structure referenced by the `geometry.instances.data` member of any element of `pGeometries` or `ppGeometries` with a `geometryType` of `VK_GEOMETRY_TYPE_INSTANCES_KHR` in any other element of `pInfos`, which is accessed by this command.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member **must** not be `VK_NULL_HANDLE`.

For each element of `pInfos`, if its `mode` member is `VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR`, its `srcAccelerationStructure` member **must** have been built before with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_KHR` set in `VkAccelerationStructureBuildGeometryInfoKHR::flags`. 
For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, its srcAccelerationStructure and dstAccelerationStructure members must either be the same VkAccelerationStructureKHR, or not have any memory aliasing.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, its geometryCount member must have the same value which was specified when srcAccelerationStructure was last built.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, its flags member must have the same value which was specified when srcAccelerationStructure was last built.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, its type member must have the same value which was specified when srcAccelerationStructure was last built.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, its geometryType member must have the same value which was specified when srcAccelerationStructure was last built.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, its flags member must have the same value which was specified when srcAccelerationStructure was last built.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its geometry.triangles.vertexFormat member must have the same value which was specified when srcAccelerationStructure was last built.

For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its geometry.triangles.maxVertex member must have the same value which was specified when srcAccelerationStructure was last built.
VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its
geometry.triangles.indexType member must have the same value which was specified
when srcAccelerationStructure was last built.

• VUID-vkBuildAccelerationStructuresKHR-pInfos-03766
  For each element of pInfo, if its mode member is
  VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
  VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
  ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, its
  geometry.triangles.transformData member was NULL when srcAccelerationStructure
  was last built, then it must be NULL.

• VUID-vkBuildAccelerationStructuresKHR-pInfos-03767
  For each element of pInfo, if its mode member is
  VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
  VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
  ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, if its
  geometry.triangles.transformData member was not NULL when srcAccelerationStructure
  was last built, then it may not be NULL.

• VUID-vkBuildAccelerationStructuresKHR-pInfos-03768
  For each element of pInfo, if its mode member is
  VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, then for each
  VkAccelerationStructureGeometryKHR structure referred to by its pGeometries or
  ppGeometries members, if geometryType is VK_GEOMETRY_TYPE_TRIANGLES_KHR, and
  geometry.triangles.indexType is not VK_INDEX_TYPE_NONE_KHR, then the value of each index
  referenced index must be the same as the corresponding index value when
  srcAccelerationStructure was last built.

• VUID-vkBuildAccelerationStructuresKHR-primitiveCount-03769
  For each VkAccelerationStructureBuildRangeInfoKHR referenced by this command, its
  primitiveCount member must have the same value which was specified when
  srcAccelerationStructure was last built.

• VUID-vkBuildAccelerationStructuresKHR-firstVertex-03770
  For each VkAccelerationStructureBuildRangeInfoKHR referenced by this command, if the
  corresponding geometry uses indices, its firstVertex member must have the same value
  which was specified when srcAccelerationStructure was last built.

• VUID-vkBuildAccelerationStructuresKHR-pInfos-03801
  For each element of pInfo[i], pGeometries or pInfo[i].ppGeometries with a geometryType of
  VK_GEOMETRY_TYPE_INSTANCES_KHR, the corresponding ppBuildRangeInfos[i][j].primitiveCount
  must be less than or equal to VkPhysicalDeviceAccelerationStructurePropertiesKHR
  ::maxInstanceCount.

• VUID-vkBuildAccelerationStructuresKHR-pInfos-03675
  For each pInfo[i], dstAccelerationStructure must have been created with a value of
  VkAccelerationStructureCreateInfoKHR::size greater than or equal to the memory size
  required by the build operation, as returned by
  vkGetAccelerationStructureBuildSizesKHR with pBuildInfo = pInfo[i] and with each
element of the pMaxPrimitiveCounts array greater than or equal to the equivalent ppBuildRangeInfos[i][j].primitiveCount values for j in [0,pInfos[i].geometryCount)

- VUID-vkBuildAccelerationStructuresKHR-ppBuildRangeInfos-03676
  Each element of ppBuildRangeInfos[i] must be a valid pointer to an array of pInfos[i].geometryCount VkAccelerationStructureBuildRangeInfoKHR structures

- VUID-vkBuildAccelerationStructuresKHR-deferredOperation-03677
  If deferredOperation is not VK_NULL_HANDLE, it must be a valid VkDeferredOperationKHR object

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03722
  For each element of pInfos, the buffer used to create its dstAccelerationStructure member must be bound to host-visible device memory

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03723
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR the buffer used to create its srcAccelerationStructure member must be bound to host-visible device memory

- VUID-vkBuildAccelerationStructuresKHR-accelerationStructureHostCommands-03581
  The VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureHostCommands feature must be enabled

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03725
  If pInfos[i].mode is VK_BUILD_ACCELERATION_STRUCTURE_MODE_BUILD_KHR, all addresses between pInfos[i].scratchData.hostAddress and pInfos[i].scratchData.hostAddress + N - 1 must be valid host memory, where N is given by the buildScratchSize member of the VkAccelerationStructureBuildSizesInfoKHR structure returned from a call to vkGetAccelerationStructureBuildSizesKHR with an identical VkAccelerationStructureBuildGeometryInfoKHR structure and primitive count

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03726
  If pInfos[i].mode is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR, all addresses between pInfos[i].scratchData.hostAddress and pInfos[i].scratchData.hostAddress + N - 1 must be valid host memory, where N is given by the updateScratchSize member of the VkAccelerationStructureBuildSizesInfoKHR structure returned from a call to vkGetAccelerationStructureBuildSizesKHR with an identical VkAccelerationStructureBuildGeometryInfoKHR structure and primitive count

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03771
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_TRIANGLES_KHR, geometry.triangles.vertexData.hostAddress must be a
valid host address

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03772
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_TRIANGLES_KHR, if geometry.triangles.indexType is not VK_INDEX_TYPE_NONE_KHR, geometry.triangles.indexData.hostAddress must be a valid host address

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03773
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_TRIANGLES_KHR, if geometry.triangles.transformData.hostAddress is not 0, it must be a valid host address

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03774
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_AABBS_KHR, geometry.aabbs.data.hostAddress must be a valid host address

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03775
  For each element of pInfos, the buffer used to create its dstAccelerationStructure member must be bound to memory that was not allocated with multiple instances

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03776
  For each element of pInfos, if its mode member is VK_BUILD_ACCELERATION_STRUCTURE_MODE_UPDATE_KHR the buffer used to create its srcAccelerationStructure member must be bound to memory that was not allocated with multiple instances

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03777
  For each element of pInfos, the buffer used to create each acceleration structure referenced by the geometry.instances.data member of any element of pGeometries or ppGeometries with a geometryType of VK_GEOMETRY_TYPE_INSTANCES_KHR must be bound to memory that was not allocated with multiple instances

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03778
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_INSTANCES_KHR, geometry.instances.data.hostAddress must be a valid host address

- VUID-vkBuildAccelerationStructuresKHR-pInfos-03779
  For any element of pInfos[i].pGeometries or pInfos[i].ppGeometries with a geometryType of VK_GEOMETRY_TYPE_INSTANCES_KHR, each VkAccelerationStructureInstanceKHR::accelerationStructureReference value in geometry.instances.data.hostAddress must be a valid VkAccelerationStructureKHR object
Valid Usage (Implicit)

- **VUID-vkBuildAccelerationStructuresKHR-device-parameter**
  
  `device` must be a valid `VkDevice` handle

- **VUID-vkBuildAccelerationStructuresKHR-deferredOperation-parameter**
  
  If `deferredOperation` is not `VK_NULL_HANDLE`, `deferredOperation` must be a valid `VkDeferredOperationKHR` handle

- **VUID-vkBuildAccelerationStructuresKHR-pInfos-parameter**
  
  `pInfos` must be a valid pointer to an array of `infoCount` valid `VkAccelerationStructureBuildGeometryInfoKHR` structures

- **VUID-vkBuildAccelerationStructuresKHR-ppBuildRangeInfos-parameter**
  
  `ppBuildRangeInfos` must be a valid pointer to an array of `infoCount` `VkAccelerationStructureBuildRangeInfoKHR` structures

- **VUID-vkBuildAccelerationStructuresKHR-infoCount-arraylength**
  
  `infoCount` must be greater than 0

- **VUID-vkBuildAccelerationStructuresKHR-deferredOperation-parent**
  
  If `deferredOperation` is a valid handle, it must have been created, allocated, or retrieved from `device`

Return Codes

**Success**

- `VK_SUCCESS`
- `VK_OPERATION_DEFERRED_KHR`
- `VK_OPERATION_NOT_DEFERRED_KHR`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

To copy or compact an acceleration structure on the host, call:

```c
// Provided by VK_KHR_acceleration_structure
VkResult vkCopyAccelerationStructureKHR(
    VkDevice                                    device,         
    VkDeferredOperationKHR                      deferredOperation, 
    const VkCopyAccelerationStructureInfoKHR*   pInfo);
```

- `device` is the device which owns the acceleration structures.
- `deferredOperation` is an optional `VkDeferredOperationKHR` to request deferral for this command.
• pInfo is a pointer to a VkCopyAccelerationStructureInfoKHR structure defining the copy operation.

This command fulfills the same task as vkCmdCopyAccelerationStructureKHR but is executed by the host.

Valid Usage

- VUID-vkCopyAccelerationStructureKHR-deferredOperation-03677
  If deferredOperation is not VK_NULL_HANDLE, it must be a valid VkDeferredOperationKHR object

- VUID-vkCopyAccelerationStructureKHR-deferredOperation-03678
  Any previous deferred operation that was associated with deferredOperation must be complete

- VUID-vkCopyAccelerationStructureKHR-buffer-03727
  The buffer used to create pInfo->src must be bound to host-visible device memory

- VUID-vkCopyAccelerationStructureKHR-buffer-03728
  The buffer used to create pInfo->dst must be bound to host-visible device memory

- VUID-vkCopyAccelerationStructureKHR-accelerationStructureHostCommands-03582
  The VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureHostCommands feature must be enabled

- VUID-vkCopyAccelerationStructureKHR-buffer-03780
  The buffer used to create pInfo->src must be bound to memory that was not allocated with multiple instances

- VUID-vkCopyAccelerationStructureKHR-buffer-03781
  The buffer used to create pInfo->dst must be bound to memory that was not allocated with multiple instances

Valid Usage (Implicit)

- VUID-vkCopyAccelerationStructureKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkCopyAccelerationStructureKHR-deferredOperation-parameter
  If deferredOperation is not VK_NULL_HANDLE, deferredOperation must be a valid VkDeferredOperationKHR handle

- VUID-vkCopyAccelerationStructureKHR-pInfo-parameter
  pInfo must be a valid pointer to a valid VkCopyAccelerationStructureInfoKHR structure

- VUID-vkCopyAccelerationStructureKHR-deferredOperation-parent
  If deferredOperation is a valid handle, it must have been created, allocated, or retrieved from device
Return Codes

**Success**

- `VK_SUCCESS`
- `VK_OPERATION_DEFERRED_KHR`
- `VK_OPERATION_NOT_DEFERRED_KHR`

**Failure**

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

To copy host accessible memory to an acceleration structure, call:

```c
// Provided by VK_KHR_acceleration_structure
VkResult vkCopyMemoryToAccelerationStructureKHR(
    VkDevice device,  // Provided by VK_KHR_acceleration_structure
    VkDeferredOperationKHR deferredOperation,
    const VkCopyMemoryToAccelerationStructureInfoKHR* pInfo);
```

- `device` is the device which owns `pInfo->dst`.
- `deferredOperation` is an optional `VkDeferredOperationKHR` to request deferral for this command.
- `pInfo` is a pointer to a `VkCopyMemoryToAccelerationStructureInfoKHR` structure defining the copy operation.

This command fulfills the same task as `vkCmdCopyMemoryToAccelerationStructureKHR` but is executed by the host.

This command can accept acceleration structures produced by either `vkCmdCopyAccelerationStructureToMemoryKHR` or `vkCopyAccelerationStructureToMemoryKHR`. 
Valid Usage

- **VUID-vkCopyMemoryToAccelerationStructureKHR-deferredOperation-03677**
  
  If `deferredOperation` is not `VK_NULL_HANDLE`, it must be a valid `VkDeferredOperationKHR` object.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-deferredOperation-03678**
  
  Any previous deferred operation that was associated with `deferredOperation` must be complete.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-pInfo-03729**
  
  `pInfo->src.hostAddress` must be a valid host pointer.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-pInfo-03750**
  
  `pInfo->src.hostAddress` must be aligned to 16 bytes.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-buffer-03730**
  
  The buffer used to create `pInfo->dst` must be bound to host-visible device memory.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-accelerationStructureHostCommands-03583**
  
  The `VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureHostCommands` feature must be enabled.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-buffer-03782**
  
  The buffer used to create `pInfo->dst` must be bound to memory that was not allocated with multiple instances.

Valid Usage (Implicit)

- **VUID-vkCopyMemoryToAccelerationStructureKHR-device-parameter**
  
  `device` must be a valid `VkDevice` handle.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-deferredOperation-parameter**
  
  If `deferredOperation` is not `VK_NULL_HANDLE`, `deferredOperation` must be a valid `VkDeferredOperationKHR` handle.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-pInfo-parameter**
  
  `pInfo` must be a valid pointer to a valid `VkCopyMemoryToAccelerationStructureInfoKHR` structure.

- **VUID-vkCopyMemoryToAccelerationStructureKHR-deferredOperation-parent**
  
  If `deferredOperation` is a valid handle, it must have been created, allocated, or retrieved from `device`.
Return Codes

Success
- VK_SUCCESS
- VK_OPERATION_DEFERRED_KHR
- VK_OPERATION_NOT_DEFERRED_KHR

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To copy an acceleration structure to host accessible memory, call:

```c
// Provided by VK_KHR_acceleration_structure
VkResult vkCopyAccelerationStructureToMemoryKHR(
    VkDevice                                    device,
    VkDeferredOperationKHR                      deferredOperation,
    const VkCopyAccelerationStructureToMemoryInfoKHR* pInfo);
```

- `device` is the device which owns `pInfo->src`.
- `deferredOperation` is an optional `VkDeferredOperationKHR` to request deferral for this command.
- `pInfo` is a pointer to a `VkCopyAccelerationStructureToMemoryInfoKHR` structure defining the copy operation.

This command fulfills the same task as `vkCmdCopyAccelerationStructureToMemoryKHR` but is executed by the host.

This command produces the same results as `vkCmdCopyAccelerationStructureToMemoryKHR`, but writes its result directly to a host pointer, and is executed on the host rather than the device. The output may not necessarily be bit-for-bit identical, but it can be equally used by either `vkCmdCopyMemoryToAccelerationStructureKHR` or `vkCopyMemoryToAccelerationStructureKHR`. 
Valid Usage

• VUID-vkCopyAccelerationStructureToMemoryKHR-deferredOperation-03677
  If deferredOperation is not VK_NULL_HANDLE, it must be a valid VkDeferredOperationKHR object

• VUID-vkCopyAccelerationStructureToMemoryKHR-deferredOperation-03678
  Any previous deferred operation that was associated with deferredOperation must be complete

• VUID-vkCopyAccelerationStructureToMemoryKHR-buffer-03731
  The buffer used to create pInfo->src must be bound to host-visible device memory

• VUID-vkCopyAccelerationStructureToMemoryKHR-pInfo-03732
  pInfo->dst.hostAddress must be a valid host pointer

• VUID-vkCopyAccelerationStructureToMemoryKHR-pInfo-03751
  pInfo->dst.hostAddress must be aligned to 16 bytes

• VUID-vkCopyAccelerationStructureToMemoryKHR-accelerationStructureHostCommands-03584
  The VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureHostCommands feature must be enabled

• VUID-vkCopyAccelerationStructureToMemoryKHR-buffer-03783
  The buffer used to create pInfo->src must be bound to memory that was not allocated with multiple instances

Valid Usage (Implicit)

• VUID-vkCopyAccelerationStructureToMemoryKHR-device-parameter
  device must be a valid VkDevice handle

• VUID-vkCopyAccelerationStructureToMemoryKHR-deferredOperation-parameter
  If deferredOperation is not VK_NULL_HANDLE, deferredOperation must be a valid VkDeferredOperationKHR handle

• VUID-vkCopyAccelerationStructureToMemoryKHR-pInfo-parameter
  pInfo must be a valid pointer to a valid VkCopyAccelerationStructureToMemoryInfoKHR structure

• VUID-vkCopyAccelerationStructureToMemoryKHR-deferredOperation-parent
  If deferredOperation is a valid handle, it must have been created, allocated, or retrieved from device
Return Codes

**Success**

- VK_SUCCESS
- VK_OPERATION_DEFERRED_KHR
- VK_OPERATION_NOT_DEFERRED_KHR

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

To query acceleration structure size parameters on the host, call:

```c
// Provided by VK_KHR_acceleration_structure
VkResult vkWriteAccelerationStructuresPropertiesKHR(
    VkDevice device,                      // device which owns the acceleration structures in pAccelerationStructures.
    uint32_t accelerationStructureCount, // count of acceleration structures for which to query the property.
    const VkAccelerationStructureKHR* pAccelerationStructures, // points to an array of existing previously built acceleration structures.
    VkQueryType queryType,               // VkQueryType value specifying the property to be queried.
    size_t dataSize,                     // size in bytes of the buffer pointed to by pData.
    void* pData,                         // pointer to a user-allocated buffer where the results will be written.
    size_t stride);                      // stride in bytes between results for individual queries within pData.
```

This command fulfills the same task as `vkCmdWriteAccelerationStructuresPropertiesKHR` but is executed by the host.
Valid Usage

- VUID-vkWriteAccelerationStructuresPropertiesKHR-accelerationStructures-03431
  All acceleration structures in `pAccelerationStructures` must have been built with `VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR` if `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-queryType-03432
  `queryType` must be `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR` or `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-queryType-03448
  If `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR`, then `stride` must be a multiple of the size of `VkDeviceSize`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-queryType-03449
  If `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR`, then `data` must point to a `VkDeviceSize`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-queryType-03450
  If `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`, then `stride` must be a multiple of the size of `VkDeviceSize`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-queryType-03451
  If `queryType` is `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`, then `data` must point to a `VkDeviceSize`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-dataSize-03452
  `dataSize` must be greater than or equal to `accelerationStructureCount*stride`

- VUID-vkWriteAccelerationStructuresPropertiesKHR-buffer-03733
  The buffer used to create each acceleration structure in `pAccelerationStructures` must be bound to host-visible device memory

- VUID-vkWriteAccelerationStructuresPropertiesKHR-accelerationStructureHostCommands-03585
  The `VkPhysicalDeviceAccelerationStructureFeaturesKHR::accelerationStructureHostCommands` feature must be enabled

- VUID-vkWriteAccelerationStructuresPropertiesKHR-buffer-03784
  The buffer used to create each acceleration structure in `pAccelerationStructures` must be bound to memory that was not allocated with multiple instances
Valid Usage (Implicit)

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-device-parameter**
  
  device must be a valid VkDevice handle

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-pAccelerationStructures-parameter**
  
  pAccelerationStructures must be a valid pointer to an array of accelerationStructureCount valid VkAccelerationStructureKHR handles

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-queryType-parameter**
  
  queryType must be a valid VkQueryType value

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-pData-parameter**
  
  pData must be a valid pointer to an array of dataSize bytes

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-accelerationStructureCount-arraylength**
  
  accelerationStructureCount must be greater than 0

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-dataSize-arraylength**
  
  dataSize must be greater than 0

- **VUID-vkWriteAccelerationStructuresPropertiesKHR-pAccelerationStructures-parent**
  
  Each element of pAccelerationStructures must have been created, allocated, or retrieved from device

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
Chapter 37. Ray Traversal

Ray traversal is the process which handles intersections between a traced ray and geometry in an acceleration structure.

Ray traversal cannot be started by a Vulkan API command directly - a shader must execute `OpRayQueryProceedKHR` or `OpTraceRayKHR`. When the `rayTracingPipeline` feature is enabled, `OpTraceRayKHR` can be used for ray tracing in a ray tracing pipeline. When the `rayQuery` feature is enabled, `OpRayQueryProceedKHR` can be used in any shader stage.

37.1. Ray Intersection Candidate Determination

Once tracing begins, rays will be tested against geometries in an acceleration structure to determine if a hit occurred between them, initially based only on their geometric properties (i.e. their vertices). The implementation performs similar operations to that of rasterization, but with the effective viewport determined by the parameters of the ray, and the geometry transformed into a space determined by that viewport.

The acceleration structure space coordinates of each primitive are transformed according to the origin and direction of the ray as follows:

\[
\begin{pmatrix}
  x_r \\
  y_r \\
  z_r 
\end{pmatrix} =
\begin{pmatrix}
  a_x^2 (1 - c) + c & a_x a_y (1 - c) - s a_z & a_x a_z (1 - c) + s a_y \\
  a_x a_y (1 - c) + s a_z & a_y^2 (1 - c) + c & a_y a_z (1 - c) - s a_x \\
  a_x a_z (1 - c) - s a_y & a_y a_z (1 - c) + s a_x & a_z^2 (1 - c) + c
\end{pmatrix}
\begin{pmatrix}
  x_{as} - o_x \\
  y_{as} - o_y \\
  z_{as} - o_z 
\end{pmatrix}
\]

\(a\) is the axis of rotation from the unnormalized ray direction vector \(d\) to the axis vector \(k\):

\[
a = \begin{pmatrix}
  d \times k \\
  \|d \times k\| \\
  0 \\\n  1 
\end{pmatrix}
\]

\(s\) and \(c\) are the sine and cosine of the angle of rotation about \(a\) from \(d\) to \(k\):

\[
s = \sqrt{1 - c^2}
\]

\(k\) is the unit vector:

\[
k = \begin{pmatrix}
  0 \\
  0 \\
  -1 
\end{pmatrix}
\]

\(o\) and \(d\) are the ray origin and unnormalized direction, respectively; the vector described by \(x_{as}, y_{as}\), and \(z_{as}\) is any position in acceleration structure space; and the vector described by \(x_r, y_r,\) and \(z_r\) is the same position in ray space.

An intersection candidate is a unique point of intersection between a ray and a geometric primitive. For any primitive that has within its bounds a position \(xyz_{as}\) such that
\[
\begin{align*}
    x_r &= 0 \\
    y_r &= 0 \\
    t_{\text{min}} &\leq \frac{z_r}{\Vert\mathbf{d}\Vert} \leq t_{\text{max}}
\end{align*}
\]

(where \( t = \frac{z_r}{\Vert\mathbf{d}\Vert} \)) an intersection candidate exists.

Figure 25. Ray intersection candidate

The determination of this condition is performed in an implementation specific manner, and may be performed with floating point operations. Due to the complexity and number of operations involved, inaccuracies are expected, particularly as the scale of values involved begins to diverge. Implementations should take efforts to maintain as much precision as possible.

Note

One very common case is when geometries are close to each other at some distance from the origin in acceleration structure space, where an effect similar to “z-fighting” is likely to be observed. Applications can mitigate this by ensuring their detailed geometries remain close to the origin.

Another likely case is when the origin of a ray is set to a position on a previously intersected surface, and its \( t_{\text{min}} \) is zero or near zero; an intersection may be detected on the emitting surface. This case can usually be mitigated by offsetting \( t_{\text{min}} \) slightly.

In the case of AABB geometries, implementations may increase their size in an acceleration structure in order to mitigate precision issues. This may result in false positives being reported to the application.

For triangle intersection candidates, the \( b \) and \( c \) barycentric coordinates on the triangle where the above condition is met are made available to future shading. If the ray was traced with \texttt{OpTraceRayKHR}, these values are available as a vector of 2 32-bit floating point values in the \texttt{HitAttributeKHR} storage class.

Once an intersection candidate is determined, it proceeds through the following operations, in order:

1. Ray Intersection Culling
2. Ray Intersection Confirmation
3. Ray Closest Hit Determination
4. Ray Result Determination
The sections below describe the exact details of these tests. There is no ordering guarantee between operations performed on different intersection candidates.

### 37.1.1. Watertightness

For a set of triangles with identical transforms, within a single instance:

- Any set of two or more triangles where all triangles have one vertex with an identical position value, that vertex is a **shared vertex**.
- Any set of two triangles with two shared vertices that were specified in the same winding order in each triangle have a **shared edge** defined by those vertices.

A *closed fan* is a set of three or more triangles where:

- All triangles in the set have the same shared vertex as one of their vertices.
- All edges that include the above vertex are shared edges.
- All above shared edges are shared by exactly two triangles from the set.
- No two triangles in the set intersect, except at shared edges.
- Every triangle in the set is joined to every other triangle in the set by a series of the above shared edges.

Implementations **should** not double-hit or miss when a ray intersects a shared edge, or a shared vertex of a closed fan.

### 37.2. Ray Intersection Culling

Candidate intersections go through several phases of culling before confirmation as an actual hit. There is no particular ordering dependency between the different culling operations.

#### 37.2.1. Ray Primitive Culling

If the `rayTraversalPrimitiveCulling` or `rayQuery` features are enabled, the `SkipTrianglesKHR` and `SkipAABBsKHR` ray flags **can** be specified when tracing a ray.

If `SkipTrianglesKHR` was included in the Ray Flags operand of the ray trace instruction, and the intersection is with a triangle primitive, the intersection is dropped, and no further processing of this intersection occurs. If `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR` was included in the pipeline, traversal with `OpTraceKHR` calls will all behave as if `SkipTrianglesKHR` was included in its Ray Flags operand.

If `SkipAABBsKHR` was included in the Ray Flags operand of the ray trace instruction, and the intersection is with an AABB primitive, the intersection is dropped, and no further processing of this intersection occurs. If `VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR` was included in the pipeline, traversal with `OpTraceKHR` calls will all behave as if `SkipAABBsKHR` was included in its Ray Flags operand.
37.2.2. Ray Mask Culling

Instances can be made invisible to particular rays based on the value of `VkAccelerationStructureInstanceKHR::mask` used to add that instance to a top-level acceleration structure, and the Cull Mask parameter used to trace the ray.

For the instance which is intersected, if `mask & Cull Mask == 0`, the intersection is dropped, and no further processing occurs.

37.2.3. Ray Face Culling

As in polygon rasterization, one of the stages of ray traversal is to determine if a triangle primitive is back- or front-facing, and primitives can be culled based on that facing.

If the intersection candidate is with an AABB primitive, this operation is skipped.

**Determination**

When a ray intersects a triangle primitive, the order that vertices are specified for the polygon affects whether the ray intersects the front or back face. Front or back facing is determined in the same way as they are for rasterization, based on the sign of the polygon's area but using the ray space coordinates instead of framebuffer coordinates. One way to compute this area is:

\[ a = -\frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1} y_i \]

where \( x_i \) and \( y_i \) are the x and y ray space coordinates of the ith vertex of the n-vertex polygon (vertices are numbered starting at zero for the purposes of this computation) and \( i \oplus 1 = (i + 1) \mod n \).

If `VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_KHR` is included in `VkAccelerationStructureInstanceKHR::flags` for the instance containing the intersected triangle, if \( a \) is positive then the intersection is with the front face of the triangle, otherwise it is with the back face. If `VK_GEOMETRY_INSTANCE_TRIANGLE_FRONT_COUNTERCLOCKWISE_BIT_KHR` was not included, this determination is reversed. Additionally, if \( a = 0 \), the intersection candidate is treated as not intersecting with any face, irrespective of the sign.

If the ray was traced with OpTraceRayKHR, the HitKindKHR built-in is set to HitKindFrontFacingTriangleKHR if the intersection is with front-facing geometry, and HitKindBackFacingTriangleKHR if the intersection is with back-facing geometry, for shader stages considering this intersection.

If the ray was traced with OpRayQueryProceedKHR, OpRayQueryGetIntersectionFrontFaceKHR will return true for intersection candidates with front faces, or false for back faces.

**Culling**

If CullBackFacingTrianglesKHR was included in the Ray Flags parameter of the ray trace instruction, and the intersection is determined as with the back face of a triangle primitive, the intersection is dropped, and no further processing of this intersection occurs.

If CullFrontFacingTrianglesKHR was included in the Ray Flags parameter of the ray trace instruction,
and the intersection is determined as with the front face of a triangle primitive, the intersection is dropped, and no further processing of this intersection occurs.

This culling is disabled if `VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR` was included in `VkAccelerationStructureInstanceKHR::flags` for the instance which the intersected geometry belongs to.

Intersection candidates that have not intersected with any face (\(a == 0\)) are unconditionally culled, irrespective of ray flags and geometry instance flags.

### 37.2.4. Ray Opacity Culling

Each geometry in the acceleration structure **may** be considered either opaque or not. Opaque geometries continue through traversal as normal, whereas non-opaque geometries need to be either confirmed or discarded by shader code. Intersection candidates **can** also be culled based on their opacity.

**Determination**

Each individual intersection candidate is initially determined as opaque if `VK_GEOMETRY_OPAQUE_BIT_KHR` was included in the `VkAccelerationStructureGeometryKHR::flags` when the geometry it intersected with was built, otherwise it is considered non-opaque.

If the intersection candidate was generated by an intersection shader, the intersection is initially considered to have opacity matching the AABB candidate that it was generated from.

However, this opacity can be overridden when it is built into an instance. Setting `VK_GEOMETRY_INSTANCE_FORCE_OPAQUE_BIT_KHR` in `VkAccelerationStructureInstanceKHR::flags` will force all geometries in the instance to be considered opaque. Similarly, setting `VK_GEOMETRY_INSTANCE_FORCE_NO_OPAQUE_BIT_KHR` will force all geometries in the instance to be considered non-opaque.

This can again be overridden by including `OpaqueKHR` or `NoOpaqueKHR` in the `Ray Flags` parameter when tracing a ray. `OpaqueKHR` forces all geometries to behave as if they are opaque, regardless of their build parameters. Similarly, `NoOpaqueKHR` forces all geometries to behave as if they are non-opaque.

If the ray was traced with `OpRayQueryProceedKHR`, to determine the opacity of AABB intersection candidates, `OpRayQueryGetIntersectionCandidateAABBOpaqueKHR` **can** be used. This instruction will return `true` for opaque intersection candidates, and `false` for non-opaque intersection candidates.

**Culling**

If `CullOpaqueKHR` is included in the `Ray Flags` parameter when tracing a ray, an intersection with a geometry that is considered opaque is dropped, and no further processing occurs.

If `CullNoOpaqueKHR` is included in the `Ray Flags` parameter when tracing a ray, an intersection with a geometry that is considered non-opaque is dropped, and no further processing occurs.
37.3. Ray Intersection Confirmation

Depending on the opacity of intersected geometry and whether it is a triangle or an AABB, candidate intersections are further processed to determine the eventual hit result. Candidates generated from AABB intersections run through the same confirmation process as triangle hits.

37.3.1. AABB Intersection Candidates

For intersection candidates with an AABB geometry generated by Ray Intersection Candidate Determination, shader code is executed to determine whether any hits should be reported to the traversal infrastructure; no further processing of this intersection candidate occurs.

If the ray was traced with OpTraceRayKHR, an intersection shader is invoked from the Shader Binding Table according to the specified indexing for the intersected geometry. If this shader calls OpReportIntersectionKHR, a new intersection candidate is generated as described below. If the intersection shader is VK_SHADER_UNUSED_KHR (which is only allowed for a zero hit group) then no intersection is generated.

Each new candidate generated as a result of this processing is a generated intersection candidate that intersects the AABB geometry, with a t value equal to the Hit parameter of the OpReportIntersectionKHR instruction. The new generated candidate is then independently run through Ray Intersection Confirmation as a generated intersection.

If the ray was traced with OpRayQueryProceedKHR, control is returned to the shader which executed OpRayQueryProceedKHR, returning true. The resulting ray query has a candidate intersection type of RayQueryCandidateIntersectionAABBKHR. OpRayQueryGenerateIntersectionKHR can be called to commit a new intersection candidate with committed intersection type of RayQueryCommittedIntersectionGeneratedKHR. Further ray query processing can be continued by executing OpRayQueryProceedKHR with the same ray query, or intersection can be terminated with OpRayQueryTerminateKHR. Unlike rays traced with OpTraceKHR, candidates generated in this way skip generated intersection candidate confirmation; applications should make this determination before generating the intersection.

This operation may be executed multiple times for the same intersection candidate.

37.3.2. Triangle and Generated Intersection Candidates

For triangle and generated intersection candidates, additional shader code may be executed based on the intersections opacity.

If the intersection is opaque, the candidate is immediately confirmed as a valid hit and passes to the next stage of processing.

For non-opaque intersection candidates, shader code is executed to determine whether a hit occurred or not.

If the ray was traced with OpTraceRayKHR, an any-hit shader is invoked from the Shader Binding Table according to the specified indexing. If this shader calls OpIgnoreIntersectionKHR, the candidate is dropped and no further processing of the candidate occurs. If the any-hit shader identified is
VK_SHADER_UNUSED_KHR, the candidate is immediately confirmed as a valid hit and passes to the next stage of processing.

If the ray was traced with OpRayQueryProceedKHR, control is returned to the shader which executed OpRayQueryProceedKHR, returning true. As only triangle candidates participate in this operation with ray queries, the resulting candidate intersection type is always RayQueryCandidateIntersectionTriangleKHR. OpRayQueryConfirmIntersectionKHR can be called on the ray query to confirm the candidate as a hit with committed intersection type of RayQueryCommittedIntersectionTriangleKHR. Further ray query processing can be continued by executing OpRayQueryProceedKHR with the same ray query, or intersection can be terminated with OpRayQueryTerminateKHR. If OpRayQueryConfirmIntersectionKHR has not been executed, the candidate is dropped and no further processing of the candidate occurs.

This operation may be executed multiple times for the same intersection candidate unless VK_GEOMETRY_NO_DUPLICATE_ANY_HIT_INVOCATION_BIT_KHR was specified for the intersected geometry.

37.4. Ray Closest Hit Determination

Unless the ray was traced with the TerminateOnFirstHitKHR ray flag, the implementation must track the closest confirmed hit until all geometries have been tested and either confirmed or dropped.

After an intersection candidate is confirmed, its t value is compared to t_max to determine which intersection is closer, where t is the parametric distance along the ray at which the intersection occurred.

- If \( t < t_{\text{max}} \), \( t_{\text{max}} \) is set to \( t \) and the candidate is set as the current closest hit.
- If \( t > t_{\text{max}} \), the candidate is dropped and no further processing of that candidate occurs.
- If \( t = t_{\text{max}} \), the candidate may be set as the current closest hit or dropped.

If TerminateOnFirstHitKHR was included in the Ray Flags used to trace the ray, once the first hit is confirmed, the ray trace is terminated.

37.5. Ray Result Determination

Once all candidates have finished processing the prior stages, or if the ray is forcibly terminated, the final result of the ray trace is determined.

If a closest hit result was identified by Ray Closest Hit Determination, a closest hit has occurred, otherwise the final result is a miss.

For rays traced with OpTraceRayKHR, if a closest hit result was identified, a closest hit shader is invoked from the Shader Binding Table according to the specified indexing for the intersected geometry. Control returns to the shader that executed OpTraceRayKHR once this shader returns. This shader is skipped if either the ray flags included SkipClosestHitShaderKHR, or if the closest hit shader identified is VK_SHADER_UNUSED_NV.

For rays traced with OpTraceRayKHR where no hit result was identified, the miss shader identified by the Miss Index parameter of OpTraceRayKHR is invoked. Control returns to the shader that executed
OpTraceRayKHR once this shader returns.

If the ray was traced with OpRayQueryProceedKHR, control is returned to the shader which executed OpRayQueryProceedKHR, returning false. If a closest hit was identified by Ray Closest Hit Determination, the ray query will now have a committed intersection type of RayQueryCommittedIntersectionGeneratedKHR or RayQueryCommittedIntersectionTriangleKHR. If no closest hit was identified, the committed intersection type will be RayQueryCommittedIntersectionNoneKHR.

No further processing of a ray query occurs after this result is determined.
Chapter 38. Ray Tracing

Ray tracing uses a separate rendering pipeline from both the graphics and compute pipelines (see Ray Tracing Pipeline).

![Ray tracing pipeline execution](image)

Caption

Interaction between the different shader stages in the ray tracing pipeline

Within the ray tracing pipeline, `OpTraceRayKHR` can be called to perform a ray traversal that invokes the various ray tracing shader stages during its execution. The relationship between the ray tracing pipeline object and the geometries present in the acceleration structure traversed is passed into the ray tracing command in a `VkBuffer` object known as a shader binding table. `OpExecuteCallableKHR` can also be used in ray tracing pipelines to invoke a callable shader.

During execution, control alternates between scheduling and other operations. The scheduling functionality is implementation-specific and is responsible for workload execution. The shader stages are programmable. **Traversal**, which refers to the process of traversing acceleration structures to find potential intersections of rays with geometry, is fixed function.

The programmable portions of the pipeline are exposed in a single-ray programming model, with each invocation handling one ray at a time. Memory operations can be synchronized using standard memory barriers. The Workgroup scope and variables with a storage class of Workgroup must not be used in the ray tracing pipeline.

### 38.1. Shader Call Instructions

A shader call is an instruction which may cause execution to continue elsewhere by creating one or more invocations that execute a different shader stage.

The shader call instructions are:

- `OpTraceRayKHR` which may invoke intersection, any-hit, closest hit, or miss shaders,
- `OpReportIntersectionKHR` which may invoke any-hit shaders, and
- `OpExecute CallableKHR` which will invoke a callable shader.
The invocations created by shader call instructions are grouped into subgroups by the implementation. Those subgroups may be unrelated to the subgroup of the parent invocation.

Trace ray instructions can be used recursively; invoked shaders can themselves execute trace ray instructions, to a maximum depth defined by the maxRecursionDepth or maxRayRecursionDepth limit.

Shaders directly invoked from the API always have a recursion depth of 0; each shader executed by a trace ray instruction has a recursion depth one higher than the recursion depth of the shader which invoked it. Applications must not invoke a shader with a recursion depth greater than the value of maxRecursionDepth or maxPipelineRayRecursionDepth specified in the pipeline.

An invocation repack instruction is a ray tracing shader call instruction where the implementation may change the set of invocations that are executing. When a repack instruction is encountered, the invocation is suspended and a new invocation begins and executes the instruction. After executing the repack instruction (which may result in other ray tracing shader stages executing) the new invocation ends and the original invocation is resumed, but it may be resumed in a different subgroup or at a different SubgroupLocalInvocationId within the same subgroup. When a subset of invocations in a subgroup execute the invocation repack instruction, those that do not execute it remain in the same subgroup at the same SubgroupLocalInvocationId.

The OpTraceRayKHR, OpReportIntersectionKHR, and OpExecuteCallableKHR instructions are invocation repack instructions.

The invocations that are executing before an invocation repack instruction, after the instruction, or are created by the instruction, are shader-call-related.

If the implementation changes the composition of subgroups, the values of SubgroupSize, SubgroupLocalInvocationId, SMIDNV, WarpIDNV, and builtin variables that are derived from them (SubgroupEqMask, SubgroupGeMask, SubgroupGtMask, SubgroupLeMask, SubgroupLtMask) must be changed accordingly by the invocation repack instruction. The application must use Volatile semantics on these Builtin variables when used in the ray generation, closest hit, miss, intersection, and callable shaders. Similarly, the application must use Volatile semantics on any RayTmaxKHR decorated Builtin used in an intersection shader.
**Note**

Subgroup operations are permitted in the programmable ray tracing shader stages. However, shader call instructions place a bound on where results of subgroup instructions or subgroup-scoped instructions that execute the dynamic instance of that instruction are potentially valid. For example, care must be taken when using the result of a ballot operation that was computed before an invocation repack instruction, after that repack instruction. The ballot may be incorrect as the set of invocations could have changed.

While the SubgroupSize built-in is required to be declared Volatile, its value will never change unless VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT is set on pipeline creation, as without that bit set, its value is required to match that of VkPhysicalDeviceSubgroupProperties::subgroupSize.

For clock operations, the value of a Subgroup scoped OpReadClockKHR read before the dynamic instance of a repack instruction should not be compared to the result of that clock instruction after the repack instruction.

When a ray tracing shader executes a dynamic instance of an invocation repack instruction which results in another ray tracing shader being invoked, their instructions are related by shader-call-order.

For ray tracing invocations that are shader-call-related:

- memory operations on StorageBuffer, Image, and ShaderRecordBufferKHR storage classes can be synchronized using the ShaderCallKHR scope.
- the CallableDataKHR, IncomingCallableDataKHR, RayPayloadKHR, HitAttributeKHR, and IncomingRayPayloadKHR storage classes are system-synchronized and no application availability and visibility operations are required.
- memory operations within a single invocation before and after the invocation repack instruction are ordered by program-order and do not require explicit synchronization.

### 38.2. Ray Tracing Commands

Ray tracing commands provoke work in the ray tracing pipeline. Ray tracing commands are recorded into a command buffer and when executed by a queue will produce work that executes according to the currently bound ray tracing pipeline. A ray tracing pipeline must be bound to a command buffer before any ray tracing commands are recorded in that command buffer.

To dispatch ray tracing use:
void vkCmdTraceRaysNV(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    raygenShaderBindingTableBuffer,
    VkDeviceSize                                raygenShaderBindingOffset,
    VkBuffer                                    missShaderBindingTableBuffer,
    VkDeviceSize                                missShaderBindingOffset,
    VkDeviceSize                                missShaderBindingStride,
    VkBuffer                                    hitShaderBindingTableBuffer,
    VkDeviceSize                                hitShaderBindingOffset,
    VkDeviceSize                                hitShaderBindingStride,
    VkBuffer                                    callableShaderBindingTableBuffer,
    VkDeviceSize                                callableShaderBindingOffset,
    VkDeviceSize                                callableShaderBindingStride,
    uint32_t                                    width,
    uint32_t                                    height,
    uint32_t                                    depth);

- **commandBuffer** is the command buffer into which the command will be recorded.
- **raygenShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the ray generation shader stage.
- **raygenShaderBindingOffset** is the offset in bytes (relative to raygenShaderBindingTableBuffer) of the ray generation shader being used for the trace.
- **missShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the miss shader stage.
- **missShaderBindingOffset** is the offset in bytes (relative to missShaderBindingTableBuffer) of the miss shader being used for the trace.
- **missShaderBindingStride** is the size in bytes of each shader binding table record in missShaderBindingTableBuffer.
- **hitShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the hit shader stages.
- **hitShaderBindingOffset** is the offset in bytes (relative to hitShaderBindingTableBuffer) of the hit shader group being used for the trace.
- **hitShaderBindingStride** is the size in bytes of each shader binding table record in hitShaderBindingTableBuffer.
- **callableShaderBindingTableBuffer** is the buffer object that holds the shader binding table data for the callable shader stage.
- **callableShaderBindingOffset** is the offset in bytes (relative to callableShaderBindingTableBuffer) of the callable shader being used for the trace.
- **callableShaderBindingStride** is the size in bytes of each shader binding table record in callableShaderBindingTableBuffer.
- **width** is the width of the ray trace query dimensions.
- **height** is height of the ray trace query dimensions.
• `depth` is depth of the ray trace query dimensions.

When the command is executed, a ray generation group of `width × height × depth` rays is assembled.
Valid Usage

- **VUID-vkCmdTraceRaysNV-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdTraceRaysNV-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- **VUID-vkCmdTraceRaysNV-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- **VUID-vkCmdTraceRaysNV-None-02692**
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- **VUID-vkCmdTraceRaysNV-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdTraceRaysNV-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- **VUID-vkCmdTraceRaysNV-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- **VUID-vkCmdTraceRaysNV-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in `Pipeline Layout Compatibility`

- **VUID-vkCmdTraceRaysNV-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in `Pipeline Layout`
Compatibility

- VUID-vkCmdTraceRaysNV-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdTraceRaysNV-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdTraceRaysNV-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdTraceRaysNV-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdTraceRaysNV-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdTraceRaysNV-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdTraceRaysNV-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdTraceRaysNV-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdTraceRaysNV-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdTraceRaysNV-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
A 

**VkPipeline** object bound to the pipeline bind point used by this command must not be a protected resource.

- **VUID-vkCmdTraceRaysNV-None-04115**
  If a **VkImageView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction must have at least as many components as the image view’s format.

- **VUID-vkCmdTraceRaysNV-OpImageWrite-04469**
  If a **VkBufferView** is accessed using **OpImageWrite** as a result of this command, then the **Type** of the **Texel** operand of that instruction must have at least as many components as the buffer view’s format.

- **VUID-vkCmdTraceRaysNV-SampledType-04470**
  If a **VkImageView** with a **VkFormat** that has a 64-bit channel width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction must have a **Width** of 64.

- **VUID-vkCmdTraceRaysNV-SampledType-04471**
  If a **VkImageView** with a **VkFormat** that has a channel width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction must have a **Width** of 32.

- **VUID-vkCmdTraceRaysNV-SampledType-04472**
  If a **VkBufferView** with a **VkFormat** that has a 64-bit channel width is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction must have a **Width** of 64.

- **VUID-vkCmdTraceRaysNV-SampledType-04473**
  If a **VkBufferView** with a **VkFormat** that has a channel width less than 64-bit is accessed as a result of this command, the **SampledType** of the **OpTypeImage** operand of that instruction must have a **Width** of 32.

- **VUID-vkCmdTraceRaysNV-sparseImageInt64Atomics-04474**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkImage** objects created with the **VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT** flag must not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdTraceRaysNV-sparseImageInt64Atomics-04475**
  If the **sparseImageInt64Atomics** feature is not enabled, **VkBuffer** objects created with the **VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT** flag must not be accessed by atomic instructions through an **OpTypeImage** with a **SampledType** with a **Width** of 64 by this command.

- **VUID-vkCmdTraceRaysNV-None-03429**
  Any shader group handle referenced by this call must have been queried from the currently bound ray tracing pipeline.

- **VUID-vkCmdTraceRaysNV-commandBuffer-04624**
  commandBuffer must not be a protected command buffer.

- **VUID-vkCmdTraceRaysNV-maxRecursionDepth-03625**
  This command must not cause a trace ray instruction to be executed from a shader invocation with a recursion depth greater than the value of **maxRecursionDepth** used to create the bound ray tracing pipeline.
If \texttt{raygenShaderBindingTableBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{raygenShaderBindingOffset} \textbf{must} be less than the size of \texttt{raygenShaderBindingTableBuffer}.

\texttt{raygenShaderBindingOffset} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupBaseAlignment}.

If \texttt{missShaderBindingTableBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{missShaderBindingOffset} \textbf{must} be less than the size of \texttt{missShaderBindingTableBuffer}.

\texttt{missShaderBindingOffset} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupBaseAlignment}.

If \texttt{hitShaderBindingTableBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{hitShaderBindingOffset} \textbf{must} be less than the size of \texttt{hitShaderBindingTableBuffer}.

\texttt{hitShaderBindingOffset} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupBaseAlignment}.

If \texttt{callableShaderBindingTableBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object.

\texttt{callableShaderBindingOffset} \textbf{must} be less than the size of \texttt{callableShaderBindingTableBuffer}.

\texttt{callableShaderBindingOffset} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupBaseAlignment}.

\texttt{missShaderBindingStride} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupHandleSize}.

\texttt{hitShaderBindingStride} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupHandleSize}.

\texttt{callableShaderBindingStride} \textbf{must} be a multiple of \texttt{VkPhysicalDeviceRayTracingPropertiesNV} \texttt{::shaderGroupHandleSize}.
- **VUID-vkCmdTraceRaysNV-missShaderBindingStride-02466**
  
  `missShaderBindingStride` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxShaderGroupStride`

- **VUID-vkCmdTraceRaysNV-hitShaderBindingStride-02467**
  
  `hitShaderBindingStride` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxShaderGroupStride`

- **VUID-vkCmdTraceRaysNV-callableShaderBindingStride-02468**
  
  `callableShaderBindingStride` **must** be less than or equal to `VkPhysicalDeviceRayTracingPropertiesNV::maxShaderGroupStride`

- **VUID-vkCmdTraceRaysNV-width-02469**
  
  `width` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`

- **VUID-vkCmdTraceRaysNV-height-02470**
  
  `height` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`

- **VUID-vkCmdTraceRaysNV-depth-02471**
  
  `depth` **must** be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`
Valid Usage (Implicit)

- VUID-vkCmdTraceRaysNV-commandBuffer-parameter: `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdTraceRaysNV-raygenShaderBindingTableBuffer-parameter: `raygenShaderBindingTableBuffer` must be a valid `VkBuffer` handle
- VUID-vkCmdTraceRaysNV-missShaderBindingTableBuffer-parameter: If `missShaderBindingTableBuffer` is not `VK_NULL_HANDLE`, `missShaderBindingTableBuffer` must be a valid `VkBuffer` handle
- VUID-vkCmdTraceRaysNV-hitShaderBindingTableBuffer-parameter: If `hitShaderBindingTableBuffer` is not `VK_NULL_HANDLE`, `hitShaderBindingTableBuffer` must be a valid `VkBuffer` handle
- VUID-vkCmdTraceRaysNV-callableShaderBindingTableBuffer-parameter: If `callableShaderBindingTableBuffer` is not `VK_NULL_HANDLE`, `callableShaderBindingTableBuffer` must be a valid `VkBuffer` handle
- VUID-vkCmdTraceRaysNV-commandBuffer-recording: `commandBuffer` must be in the recording state
- VUID-vkCmdTraceRaysNV-commandBuffer-cmdpool: The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations
- VUID-vkCmdTraceRaysNV-renderpass: This command must only be called outside of a render pass instance
- VUID-vkCmdTraceRaysNV-commonparent: Each of `callableShaderBindingTableBuffer`, `commandBuffer`, `hitShaderBindingTableBuffer`, `missShaderBindingTableBuffer`, and `raygenShaderBindingTableBuffer` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To dispatch ray tracing use:

```c
// Provided by VK_KHR_ray_tracing_pipeline
void vkCmdTraceRaysKHR(
    VkCommandBuffer                             commandBuffer,
    const VkStridedDeviceAddressRegionKHR*      pRaygenShaderBindingTable,
    const VkStridedDeviceAddressRegionKHR*      pMissShaderBindingTable,
    const VkStridedDeviceAddressRegionKHR*      pHitShaderBindingTable,
    const VkStridedDeviceAddressRegionKHR*      pCallableShaderBindingTable,
    uint32_t                                    width,
    uint32_t                                    height,
    uint32_t                                    depth);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `pRaygenShaderBindingTable` is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the ray generation shader stage.
- `pMissShaderBindingTable` is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the miss shader stage.
- `pHitShaderBindingTable` is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the hit shader stage.
- `pCallableShaderBindingTable` is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the callable shader stage.
- `width` is the width of the ray trace query dimensions.
- `height` is height of the ray trace query dimensions.
- `depth` is depth of the ray trace query dimensions.

When the command is executed, a ray generation group of \( \text{width} \times \text{height} \times \text{depth} \) rays is assembled.
Valid Usage

- **VUID-vkCmdTraceRaysKHR-magFilter-04553**
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdTraceRaysKHR-mipmapMode-04770**
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- **VUID-vkCmdTraceRaysKHR-None-02691**
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features **must** contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`.

- **VUID-vkCmdTraceRaysKHR-filterCubic-02694**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdTraceRaysKHR-filterCubicMinmax-02695**
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command **must** have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`.

- **VUID-vkCmdTraceRaysKHR-flags-02696**
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command **must** only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`.

- **VUID-vkCmdTraceRaysKHR-None-02697**
  For each set `n` that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set **must** have been bound to `n` at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set `n`, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.

- **VUID-vkCmdTraceRaysKHR-None-02698**
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value **must** have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility.
Compatibility

- VUID-vkCmdTraceRaysKHR-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, must be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdTraceRaysKHR-None-02700
  A valid pipeline must be bound to the pipeline bind point used by this command.

- VUID-vkCmdTraceRaysKHR-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdTraceRaysKHR-None-02859
  There must not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdTraceRaysKHR-None-02702
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdTraceRaysKHR-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdTraceRaysKHR-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a LOD bias or any offset values, in any shader stage.

- VUID-vkCmdTraceRaysKHR-None-02705
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdTraceRaysKHR-None-02706
  If the robust buffer access feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it must not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdTraceRaysKHR-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the
VkPipeline object bound to the pipeline bind point used by this command must not be a protected resource

- VUID-vkCmdTraceRaysKHR-None-04115
  If a VkImageView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- VUID-vkCmdTraceRaysKHR-OpImageWrite-04469
  If a VkBufferView is accessed using OpImageWrite as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- VUID-vkCmdTraceRaysKHR-SampledType-04470
  If a VkImageView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdTraceRaysKHR-SampledType-04471
  If a VkImageView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdTraceRaysKHR-SampledType-04472
  If a VkBufferView with a VkFormat that has a 64-bit channel width is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 64.

- VUID-vkCmdTraceRaysKHR-SampledType-04473
  If a VkBufferView with a VkFormat that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the OpTypeImage operand of that instruction must have a Width of 32.

- VUID-vkCmdTraceRaysKHR-sparseImageInt64Atomics-04474
  If the sparseImageInt64Atomics feature is not enabled, VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdTraceRaysKHR-sparseImageInt64Atomics-04475
  If the sparseImageInt64Atomics feature is not enabled, VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT flag must not be accessed by atomic instructions through an OpTypeImage with a SampledType with a Width of 64 by this command.

- VUID-vkCmdTraceRaysKHR-None-03429
  Any shader group handle referenced by this call must have been queried from the currently bound ray tracing pipeline

- VUID-vkCmdTraceRaysKHR-maxPipelineRayRecursionDepth-03679
  This command must not cause a shader call instruction to be executed from a shader invocation with a recursion depth greater than the value of maxPipelineRayRecursionDepth used to create the bound ray tracing pipeline

- VUID-vkCmdTraceRaysKHR-pRayGenShaderBindingTable-03680
  If the buffer from which pRayGenShaderBindingTable->deviceAddress was queried is non-
sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdTraceRaysKHR-pRayGenShaderBindingTable-03681**
  The buffer from which the `pRayGenShaderBindingTable->deviceAddress` is queried **must** have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag

- **VUID-vkCmdTraceRaysKHR-pRayGenShaderBindingTable-03682**
  `pRayGenShaderBindingTable->deviceAddress` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`

- **VUID-vkCmdTraceRaysKHR-size-04023**
  The size member of `pRayGenShaderBindingTable` **must** be equal to its stride member

- **VUID-vkCmdTraceRaysKHR-pMissShaderBindingTable-03683**
  If the buffer from which `pMissShaderBindingTable->deviceAddress` was queried is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdTraceRaysKHR-pMissShaderBindingTable-03684**
  The buffer from which the `pMissShaderBindingTable->deviceAddress` is queried **must** have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag

- **VUID-vkCmdTraceRaysKHR-pMissShaderBindingTable-03685**
  `pMissShaderBindingTable->deviceAddress` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`

- **VUID-vkCmdTraceRaysKHR-stride-03686**
  The stride member of `pMissShaderBindingTable` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleAlignment`

- **VUID-vkCmdTraceRaysKHR-stride-04029**
  The stride member of `pMissShaderBindingTable` **must** be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxShaderGroupStride`

- **VUID-vkCmdTraceRaysKHR-pHitShaderBindingTable-03687**
  If the buffer from which `pHitShaderBindingTable->deviceAddress` was queried is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdTraceRaysKHR-pHitShaderBindingTable-03688**
  The buffer from which the `pHitShaderBindingTable->deviceAddress` is queried **must** have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag

- **VUID-vkCmdTraceRaysKHR-pHitShaderBindingTable-03689**
  `pHitShaderBindingTable->deviceAddress` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`

- **VUID-vkCmdTraceRaysKHR-stride-03690**
  The stride member of `pHitShaderBindingTable` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleAlignment`

- **VUID-vkCmdTraceRaysKHR-stride-04035**
  The stride member of `pHitShaderBindingTable` **must** be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxShaderGroupStride`
If the buffer from which `pCallableShaderBindingTable->deviceAddress` was queried is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

The buffer from which the `pCallableShaderBindingTable->deviceAddress` is queried must have been created with the VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR usage flag.

`pCallableShaderBindingTable->deviceAddress` must be a multiple of VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment.

The stride member of `pCallableShaderBindingTable` must be a multiple of VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleAlignment.

The stride member of `pCallableShaderBindingTable` must be less than or equal to VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxShaderGroupStride.

If the currently bound ray tracing pipeline was created with flags that included VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR, the deviceAddress member of `pHitShaderBindingTable` must not be zero.

If the currently bound ray tracing pipeline was created with flags that included VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR, the deviceAddress member of `pHitShaderBindingTable` must not be zero.

If the currently bound ray tracing pipeline was created with flags that included VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR, the shader group handle identified by `pMissShaderBindingTable` must contain a valid miss shader.

If the currently bound ray tracing pipeline was created with flags that included VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR, entries in `pHitShaderBindingTable` accessed as a result of this command in order to execute an any-hit shader must not be set to zero.

If the currently bound ray tracing pipeline was created with flags that included VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR, entries in `pHitShaderBindingTable` accessed as a result of this command in order to execute a closest hit shader must not be set to zero.

If the currently bound ray tracing pipeline was created with flags that included VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR, entries in `pHitShaderBindingTable` accessed as a result of this command in order to execute an intersection shader must not be set to zero.
Any non-zero hit group entries in `pHitShaderBindingTable` accessed by this call from a geometry with a `geometryType` of `VK_GEOMETRY_TYPE_TRIANGLES_KHR` must have been created with `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR`.

Any non-zero hit group entries in `pHitShaderBindingTable` accessed by this call from a geometry with a `geometryType` of `VK_GEOMETRY_TYPE_AABBS_KHR` must have been created with `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`.

`commandBuffer` must not be a protected command buffer.

`width` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0] × VkPhysicalDeviceLimits::maxComputeWorkGroupSize[0]`.

`height` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1] × VkPhysicalDeviceLimits::maxComputeWorkGroupSize[1]`.


`width × height × depth` must be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxRayDispatchInvocationCount`.
Valid Usage (Implicit)

- **VUID-vkCmdTraceRaysKHR-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdTraceRaysKHR-pRaygenShaderBindingTable-parameter**
  
  `pRaygenShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure.

- **VUID-vkCmdTraceRaysKHR-pMissShaderBindingTable-parameter**
  
  `pMissShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure.

- **VUID-vkCmdTraceRaysKHR-pHitShaderBindingTable-parameter**
  
  `pHitShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure.

- **VUID-vkCmdTraceRaysKHR-pCallableShaderBindingTable-parameter**
  
  `pCallableShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure.

- **VUID-vkCmdTraceRaysKHR-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdTraceRaysKHR-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.

- **VUID-vkCmdTraceRaysKHR-renderpass**
  
  This command must only be called outside of a render pass instance.

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkStridedDeviceAddressRegionKHR` structure is defined as:
// Provided by VK_KHR_ray_tracing_pipeline
typedef struct VkStridedDeviceAddressRegionKHR {
    VkDeviceAddress    deviceAddress;
    VkDeviceSize       stride;
    VkDeviceSize       size;
} VkStridedDeviceAddressRegionKHR;

- **deviceAddress** is the device address (as returned by the `vkGetBufferDeviceAddress` command) at which the region starts, or zero if the region is unused.
- **stride** is the byte stride between consecutive elements.
- **size** is the size in bytes of the region starting at `deviceAddress`.

### Valid Usage

- **VUID-VkStridedDeviceAddressRegionKHR-size-04631**
  If `size` is not zero, all addresses between `deviceAddress` and `deviceAddress + size - 1` must be in the buffer device address range of the same buffer.

- **VUID-VkStridedDeviceAddressRegionKHR-size-04632**
  If `size` is not zero, `stride` must be less than the size of the buffer from which `deviceAddress` was queried.

To dispatch ray tracing, with some parameters sourced on the device, use:

// Provided by VK_KHR_ray_tracing_pipeline
void vkCmdTraceRaysIndirectKHR(
    VkCommandBuffer commandBuffer,
    const VkStridedDeviceAddressRegionKHR* pRaygenShaderBindingTable,
    const VkStridedDeviceAddressRegionKHR* pMissShaderBindingTable,
    const VkStridedDeviceAddressRegionKHR* pHitShaderBindingTable,
    const VkStridedDeviceAddressRegionKHR* pCallableShaderBindingTable,
    VkDeviceAddress indirectDeviceAddress);

- **commandBuffer** is the command buffer into which the command will be recorded.
- **pRaygenShaderBindingTable** is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the ray generation shader stage.
- **pMissShaderBindingTable** is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the miss shader stage.
- **pHitShaderBindingTable** is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the hit shader stage.
- **pCallableShaderBindingTable** is a `VkStridedDeviceAddressRegionKHR` that holds the shader binding table data for the callable shader stage.
- **indirectDeviceAddress** is a buffer device address which points to a `VkTraceRaysIndirectCommandKHR` structure which contains the trace ray parameters.
vkCmdTraceRaysIndirectKHR behaves similarly to vkCmdTraceRaysKHR except that the ray trace query dimensions are read by the device from indirectDeviceAddress during execution.
Valid Usage

- VUID-vkCmdTraceRaysIndirectKHR-magFilter-04553
  If a `VkSampler` created with `magFilter` or `minFilter` equal to `VK_FILTER_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- VUID-vkCmdTraceRaysIndirectKHR-mipmapMode-04770
  If a `VkSampler` created with `mipmapMode` equal to `VK_SAMPLER_MIPMAP_MODE_LINEAR` and `compareEnable` equal to `VK_FALSE` is used to sample a `VkImageView` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`

- VUID-vkCmdTraceRaysIndirectKHR-None-02691
  If a `VkImageView` is accessed using atomic operations as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`

- VUID-vkCmdTraceRaysIndirectKHR-None-02692
  If a `VkImageView` is sampled with `VK_FILTER_CUBIC_EXT` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT`

- VUID-vkCmdTraceRaysIndirectKHR-filterCubic-02694
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubic` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- VUID-vkCmdTraceRaysIndirectKHR-filterCubicMinmax-02695
  Any `VkImageView` being sampled with `VK_FILTER_CUBIC_EXT` with a reduction mode of either `VK_SAMPLER_REDUCTION_MODE_MIN` or `VK_SAMPLER_REDUCTION_MODE_MAX` as a result of this command must have a `VkImageViewType` and format that supports cubic filtering together with minmax filtering, as specified by `VkFilterCubicImageViewImageFormatPropertiesEXT::filterCubicMinmax` returned by `vkGetPhysicalDeviceImageFormatProperties2`

- VUID-vkCmdTraceRaysIndirectKHR-flags-02696
  Any `VkImage` created with a `VkImageCreateInfo::flags` containing `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` sampled as a result of this command must only be sampled using a `VkSamplerAddressMode` of `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE`

- VUID-vkCmdTraceRaysIndirectKHR-None-02697
  For each set n that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a descriptor set must have been bound to n at the same pipeline bind point, with a `VkPipelineLayout` that is compatible for set n, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout Compatibility

- VUID-vkCmdTraceRaysIndirectKHR-None-02698
  For each push constant that is statically used by the `VkPipeline` bound to the pipeline bind point used by this command, a push constant value must have been set for the same pipeline bind point, with a `VkPipelineLayout` that is compatible for push constants, with the `VkPipelineLayout` used to create the current `VkPipeline`, as described in Pipeline Layout
Compatibility

- VUID-vkCmdTraceRaysIndirectKHR-None-02699
  Descriptors in each bound descriptor set, specified via `vkCmdBindDescriptorSets`, **must** be valid if they are statically used by the `VkPipeline` bound to the pipeline bind point used by this command.

- VUID-vkCmdTraceRaysIndirectKHR-None-02700
  A valid pipeline **must** be bound to the pipeline bind point used by this command.

- VUID-vkCmdTraceRaysIndirectKHR-commandBuffer-02701
  If the `VkPipeline` object bound to the pipeline bind point used by this command requires any dynamic state, that state **must** have been set or inherited for `commandBuffer`, and done so after any previously bound pipeline with the corresponding state not specified as dynamic.

- VUID-vkCmdTraceRaysIndirectKHR-None-02702
  There **must** not have been any calls to dynamic state setting commands for any state not specified as dynamic in the `VkPipeline` object bound to the pipeline bind point used by this command, since that pipeline was bound.

- VUID-vkCmdTraceRaysIndirectKHR-None-02703
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used to sample from any `VkImage` with a `VkImageView` of the type `VK_IMAGE_VIEW_TYPE_3D`, `VK_IMAGE_VIEW_TYPE_CUBE`, `VK_IMAGE_VIEW_TYPE_1D_ARRAY`, `VK_IMAGE_VIEW_TYPE_2D_ARRAY` or `VK_IMAGE_VIEW_TYPE_CUBE_ARRAY`, in any shader stage.

- VUID-vkCmdTraceRaysIndirectKHR-None-02704
  If the `VkPipeline` object bound to the pipeline bind point used by this command accesses a `VkSampler` object that uses unnormalized coordinates, that sampler **must** not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with `ImplicitLod`, `Dref` or `Proj` in their name, in any shader stage.

- VUID-vkCmdTraceRaysIndirectKHR-None-02705
  If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a uniform buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdTraceRaysIndirectKHR-None-02706
  If the `robust buffer access` feature is not enabled, and if the `VkPipeline` object bound to the pipeline bind point used by this command accesses a storage buffer, it **must** not access values outside of the range of the buffer as specified in the descriptor set bound to the same pipeline bind point.

- VUID-vkCmdTraceRaysIndirectKHR-commandBuffer-02707
  If `commandBuffer` is an unprotected command buffer, any resource accessed by the...
A `VkPipeline` object bound to the pipeline bind point used by this command must not be a protected resource.

- **VUID-vkCmdTraceRaysIndirectKHR-None-04115**
  If a `VkImageView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the image view’s format.

- **VUID-vkCmdTraceRaysIndirectKHR-OpImageWrite-04469**
  If a `VkBufferView` is accessed using `OpImageWrite` as a result of this command, then the Type of the Texel operand of that instruction must have at least as many components as the buffer view’s format.

- **VUID-vkCmdTraceRaysIndirectKHR-SampledType-04470**
  If a `VkImageView` with a `VkFormat` that has a 64-bit channel width is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 64.

- **VUID-vkCmdTraceRaysIndirectKHR-SampledType-04471**
  If a `VkImageView` with a `VkFormat` that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 32.

- **VUID-vkCmdTraceRaysIndirectKHR-SampledType-04472**
  If a `VkBufferView` with a `VkFormat` that has a 64-bit channel width is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 64.

- **VUID-vkCmdTraceRaysIndirectKHR-SampledType-04473**
  If a `VkBufferView` with a `VkFormat` that has a channel width less than 64-bit is accessed as a result of this command, the SampledType of the `OpTypeImage` operand of that instruction must have a Width of 32.

- **VUID-vkCmdTraceRaysIndirectKHR-sparseImageInt64Atomics-04474**
  If the `sparseImageInt64Atomics` feature is not enabled, `VkImage` objects created with the `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a SampledType with a Width of 64 by this command.

- **VUID-vkCmdTraceRaysIndirectKHR-sparseImageInt64Atomics-04475**
  If the `sparseImageInt64Atomics` feature is not enabled, `VkBuffer` objects created with the `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT` flag must not be accessed by atomic instructions through an `OpTypeImage` with a SampledType with a Width of 64 by this command.

- **VUID-vkCmdTraceRaysIndirectKHR-None-03429**
  Any shader group handle referenced by this call must have been queried from the currently bound ray tracing pipeline.

- **VUID-vkCmdTraceRaysIndirectKHR-maxPipelineRayRecursionDepth-03679**
  This command must not cause a shader call instruction to be executed from a shader invocation with a recursion depth greater than the value of `maxPipelineRayRecursionDepth` used to create the bound ray tracing pipeline.

- **VUID-vkCmdTraceRaysIndirectKHR-pRayGenShaderBindingTable-03680**
  If the buffer from which `pRayGenShaderBindingTable->deviceAddress` was queried is non-
sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdTraceRaysIndirectKHR-pRayGenShaderBindingTable-03681**
  The buffer from which the `pRayGenShaderBindingTable->deviceAddress` is queried must have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag

- **VUID-vkCmdTraceRaysIndirectKHR-pRayGenShaderBindingTable-03682**
  `pRayGenShaderBindingTable->deviceAddress` must be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`

- **VUID-vkCmdTraceRaysIndirectKHR-size-04023**
  The `size` member of `pRayGenShaderBindingTable` must be equal to its `stride` member

- **VUID-vkCmdTraceRaysIndirectKHR-pMissShaderBindingTable-03683**
  If the buffer from which `pMissShaderBindingTable->deviceAddress` was queried is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdTraceRaysIndirectKHR-pMissShaderBindingTable-03684**
  The buffer from which the `pMissShaderBindingTable->deviceAddress` is queried must have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag

- **VUID-vkCmdTraceRaysIndirectKHR-pMissShaderBindingTable-03685**
  `pMissShaderBindingTable->deviceAddress` must be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`

- **VUID-vkCmdTraceRaysIndirectKHR-stride-03686**
  The `stride` member of `pMissShaderBindingTable` must be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleAlignment`

- **VUID-vkCmdTraceRaysIndirectKHR-stride-04029**
  The `stride` member of `pMissShaderBindingTable` must be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxShaderGroupStride`

- **VUID-vkCmdTraceRaysIndirectKHR-pHitShaderBindingTable-03687**
  If the buffer from which `pHitShaderBindingTable->deviceAddress` was queried is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object

- **VUID-vkCmdTraceRaysIndirectKHR-pHitShaderBindingTable-03688**
  The buffer from which the `pHitShaderBindingTable->deviceAddress` is queried must have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag

- **VUID-vkCmdTraceRaysIndirectKHR-pHitShaderBindingTable-03689**
  `pHitShaderBindingTable->deviceAddress` must be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`

- **VUID-vkCmdTraceRaysIndirectKHR-stride-03690**
  The `stride` member of `pHitShaderBindingTable` must be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleAlignment`

- **VUID-vkCmdTraceRaysIndirectKHR-stride-04035**
  The `stride` member of `pHitShaderBindingTable` must be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxShaderGroupStride`
If the buffer from which `pCallableShaderBindingTable->deviceAddress` was queried is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.

The buffer from which the `pCallableShaderBindingTable->deviceAddress` is queried **must** have been created with the `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` usage flag.

`pCallableShaderBindingTable->deviceAddress` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupBaseAlignment`.

The stride member of `pCallableShaderBindingTable` **must** be a multiple of `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleAlignment`.

The stride member of `pCallableShaderBindingTable` **must** be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxShaderGroupStride`.

If the currently bound ray tracing pipeline was created with `flags` that included `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR`, the `deviceAddress` member of `pHitShaderBindingTable` **must** not be zero.

If the currently bound ray tracing pipeline was created with `flags` that included `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR`, the `deviceAddress` member of `pHitShaderBindingTable` **must** not be zero.

If the currently bound ray tracing pipeline was created with `flags` that included `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR`, the shader group handle identified by `pMissShaderBindingTable` **must** contain a valid miss shader.

If the currently bound ray tracing pipeline was created with `flags` that included `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR`, entries in `pHitShaderBindingTable` accessed as a result of this command in order to execute an any-hit shader **must** not be set to zero.

If the currently bound ray tracing pipeline was created with `flags` that included `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR`, entries in `pHitShaderBindingTable` accessed as a result of this command in order to execute a closest hit shader **must** not be set to zero.

If the currently bound ray tracing pipeline was created with `flags` that included `VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR`, entries in `pHitShaderBindingTable` accessed as a result of this command in order to execute an intersection shader **must** not be set to zero.
Any non-zero hit group entries in `pHitShaderBindingTable` accessed by this call from a geometry with a `geometryType` of `VK_GEOMETRY_TYPE_TRIANGLES_KHR` must have been created with `VK_RAY_TRACING_SHADER_GROUP_TYPE_TRIANGLES_HIT_GROUP_KHR`.

Any non-zero hit group entries in `pHitShaderBindingTable` accessed by this call from a geometry with a `geometryType` of `VK_GEOMETRY_TYPE_AABBS_KHR` must have been created with `VK_RAY_TRACING_SHADER_GROUP_TYPE_PROCEDURAL_HIT_GROUP_KHR`.

If the buffer from which `indirectDeviceAddress` was queried is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.

The buffer from which `indirectDeviceAddress` was queried must have been created with the `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT` bit set.

`indirectDeviceAddress` must be a multiple of 4.

`commandBuffer` must not be a protected command buffer.

All device addresses between `indirectDeviceAddress` and `indirectDeviceAddress + sizeof(VkTraceRaysIndirectCommandKHR) - 1` must be in the buffer device address range of the same buffer.

The `VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipelineTraceRaysIndirect` feature must be enabled.
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pRaygenShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure
- `pMissShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure
- `pHitShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure
- `pCallableShaderBindingTable` must be a valid pointer to a valid `VkStridedDeviceAddressRegionKHR` structure
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support compute operations
- This command must only be called outside of a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `VkTraceRaysIndirectCommandKHR` structure is defined as:
typedef struct VkTraceRaysIndirectCommandKHR {
    uint32_t width;
    uint32_t height;
    uint32_t depth;
} VkTraceRaysIndirectCommandKHR;

- **width** is the width of the ray trace query dimensions.
- **height** is height of the ray trace query dimensions.
- **depth** is depth of the ray trace query dimensions.

The members of `VkTraceRaysIndirectCommandKHR` have the same meaning as the similarly named parameters of `vkCmdTraceRaysKHR`.

### Valid Usage

- **VUID-VkTraceRaysIndirectCommandKHR-width-03638**
  
  `width` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0] × VkPhysicalDeviceLimits::maxComputeWorkGroupSize[0]`

- **VUID-VkTraceRaysIndirectCommandKHR-height-03639**
  
  `height` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1] × VkPhysicalDeviceLimits::maxComputeWorkGroupSize[1]`

- **VUID-VkTraceRaysIndirectCommandKHR-depth-03640**
  
  `depth` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2] × VkPhysicalDeviceLimits::maxComputeWorkGroupSize[2]`

- **VUID-VkTraceRaysIndirectCommandKHR-width-03641**
  
  `width × height × depth` must be less than or equal to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxRayDispatchInvocationCount`

### 38.3. Shader Binding Table

A *shader binding table* is a resource which establishes the relationship between the ray tracing pipeline and the acceleration structures that were built for the ray tracing pipeline. It indicates the shaders that operate on each geometry in an acceleration structure. In addition, it contains the resources accessed by each shader, including indices of textures, buffer device addresses, and constants. The application allocates and manages shader binding tables as `VkBuffer` objects.

Each entry in the shader binding table consists of shaderGroupHandleSize bytes of data, either as queried by `vkGetRayTracingShaderGroupHandlesKHR` to refer to those specified shaders, or all zeros to refer to a zero hit group. A zero hit group behaves as though it is a hit group consisting entirely of `VK_SHADER_UNUSED_KHR`. The remainder of the data specified by the stride is application-visible data that can be referenced by a ShaderRecordBufferKHR block in the shader.

The shader binding tables to use in a ray tracing pipeline are passed to the `vkCmdTraceRaysNV`,
vkCmdTraceRaysKHR, or vkCmdTraceRaysIndirectKHR commands. Shader binding tables are read-only in shaders that are executing on the ray tracing pipeline.

Shader variables identified with the ShaderRecordBufferKHR storage class are used to access the provided shader binding table. Such variables must be:

- typed as OpTypeStruct, or an array of this type,
- identified with a Block decoration, and
- laid out explicitly using the Offset, ArrayStride, and MatrixStride decorations as specified in Offset and Stride Assignment.

The Offset decoration for any member of a Block-decorated variable in the ShaderRecordBufferKHR storage class must not cause the space required for that variable to extend outside the range [0, maxStorageBufferRange).

Accesses to the shader binding table from ray tracing pipelines must be synchronized with the VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR pipeline stage and an access type of VK_ACCESS_SHADER_READ_BIT.

### 38.3.1. Indexing Rules

In order to execute the correct shaders and access the correct resources during a ray tracing dispatch, the implementation must be able to locate shader binding table entries at various stages of execution. This is accomplished by defining a set of indexing rules that compute shader binding table record positions relative to the buffer's base address in memory. The application must organize the contents of the shader binding table's memory in a way that application of the indexing rules will lead to correct records.

**Ray Generation Shaders**

Only one ray generation shader is executed per ray tracing dispatch.

For vkCmdTraceRaysKHR, the location of the ray generation shader is specified by the pRaygenShaderBindingTable->deviceAddress parameter — there is no indexing. All data accessed must be less than pRaygenShaderBindingTable->size bytes from deviceAddress. pRaygenShaderBindingTable->stride is unused, and must be equal to pRaygenShaderBindingTable->size.

For vkCmdTraceRaysNV, the location of the ray generation shader is specified by the raygenShaderBindingTableBuffer and raygenShaderBindingOffset parameters — there is no indexing.

**Hit Shaders**

The base for the computation of intersection, any-hit, and closest hit shader locations is the instanceShaderBindingTableRecordOffset value stored with each instance of a top-level acceleration structure (VkAccelerationStructureInstanceKHR). This value determines the beginning of the shader binding table records for a given instance.

In the following rule, geometryIndex refers to the geometry index of the intersected geometry within
The `sbtRecordOffset` and `sbtRecordStride` values are passed in as parameters to `traceNV()` or `traceRayEXT()` calls made in the shaders. See Section 8.19 (Ray Tracing Functions) of the OpenGL Shading Language Specification for more details. In SPIR-V, these correspond to the `SBTOffset` and `SBTStride` parameters to the `OpTraceRayNV` or `OpTraceRayKHR` instruction.

The result of this computation is then added to `pHitShaderBindingTable->deviceAddress`, a device address passed to `vkCmdTraceRaysKHR`, or `hitShaderBindingOffset`, a base offset passed to `vkCmdTraceRaysNV`.

For `vkCmdTraceRaysKHR`, the complete rule to compute a hit shader binding table record address in the `pHitShaderBindingTable` is:

\[
\text{pHitShaderBindingTable->deviceAddress} + \text{pHitShaderBindingTable->stride} \times (\text{instanceShaderBindingTableRecordOffset} + \text{geometryIndex} \times \text{sbtRecordStride} + \text{sbtRecordOffset})
\]

All data accessed must be less than `pHitShaderBindingTable->size` bytes from the base address.

For `vkCmdTraceRaysNV`, the offset and stride come from direct parameters, so the full rule to compute a hit shader binding table record address in the `hitShaderBindingTableBuffer` is:

\[
\text{hitShaderBindingOffset} + \text{hitShaderBindingStride} \times (\text{instanceShaderBindingTableRecordOffset} + \text{geometryIndex} \times \text{sbtRecordStride} + \text{sbtRecordOffset})
\]

**Miss Shaders**

A miss shader is executed whenever a ray query fails to find an intersection for the given scene geometry. Multiple miss shaders may be executed throughout a ray tracing dispatch.

The base for the computation of miss shader locations is `pMissShaderBindingTable->deviceAddress`, a device address passed into `vkCmdTraceRaysKHR`, or `missShaderBindingOffset`, a base offset passed into `vkCmdTraceRaysNV`.

The `missIndex` value is passed in as a parameter to `traceNV()` or `traceRayEXT()` calls made in the shaders. See Section 8.19 (Ray Tracing Functions) of the OpenGL Shading Language Specification for more details. In SPIR-V, this corresponds to the `MissIndex` parameter to the `OpTraceRayNV` or `OpTraceRayKHR` instruction.

For `vkCmdTraceRaysKHR`, the complete rule to compute a miss shader binding table record address in the `pMissShaderBindingTable` is:

\[
\text{pMissShaderBindingTable->deviceAddress} + \text{pMissShaderBindingTable->stride} \times \text{missIndex}
\]

All data accessed must be less than `pMissShaderBindingTable->size` bytes from the base address.

For `vkCmdTraceRaysNV`, the offset and stride come from direct parameters, so the full rule to
compute a miss shader binding table record address in the missShaderBindingTableBuffer is:

\[
\text{missShaderBindingOffset} + \text{missShaderBindingStride} \times \text{missIndex}
\]

**Callable Shaders**

A callable shader is executed when requested by a ray tracing shader. Multiple callable shaders may be executed throughout a ray tracing dispatch.

The base for the computation of callable shader locations is pCallableShaderBindingTable->deviceAddress, a device address passed into vkCmdTraceRaysKHR, or callableShaderBindingOffset, a base offset passed into vkCmdTraceRaysNV.

The sbtRecordIndex value is passed in as a parameter to executeCallableNV() or executeCallableEXT() calls made in the shaders. See Section 8.19 (Ray Tracing Functions) of the OpenGL Shading Language Specification for more details. In SPIR-V, this corresponds to the SBTIndex parameter to the OpExecuteCallableNV or OpExecuteCallableKHR instruction.

For vkCmdTraceRaysKHR, the complete rule to compute a callable shader binding table record address in the pCallableShaderBindingTable is:

\[
p\text{CallableShaderBindingTable}->\text{deviceAddress} + p\text{CallableShaderBindingTable}->\text{stride} \times \text{sbtRecordIndex}
\]

All data accessed must be less than pCallableShaderBindingTable->size bytes from the base address.

For vkCmdTraceRaysNV, the offset and stride come from direct parameters, so the full rule to compute a callable shader binding table record address in the callableShaderBindingTableBuffer is:

\[
\text{callableShaderBindingOffset} + \text{callableShaderBindingStride} \times \text{sbtRecordIndex}
\]

**38.4. Ray Tracing Pipeline Stack**

Ray tracing pipelines have a potentially large set of shaders which may be invoked in various call chain combinations to perform ray tracing. To store parameters for a given shader execution, an implementation may use a stack of data in memory. This stack must be sized to the sum of the stack sizes of all shaders in any call chain executed by the application.

If the stack size is not set explicitly, the stack size for a pipeline is:

\[
\text{rayGenStackMax} + \min(1, \text{maxPipelineRayRecursionDepth}) \times \max(\text{closestHitStackMax}, \text{missStackMax, intersectionStackMax + anyHitStackMax}) + \max(0, \text{maxPipelineRayRecursionDepth} - 1) \times \max(\text{closestHitStackMax, missStackMax}) + 2 \times \text{callableStackMax}
\]

where rayGenStackMax, closestHitStackMax, missStackMax, intersectionStackMax, and callableStackMax are the maximum stack values queried by the
respective shader stages for any shaders in any shader groups defined by the pipeline.

This stack size is potentially significant, so an application may want to provide a more optimal stack size after pipeline compilation. The value that the application provides is the maximum value of the sum of all shaders in a call chain across all possible call chains, taking into account any application specific knowledge about the properties of the call chains.

Note
For example, if an implementation has two types of closest hit and miss shaders that it can use but that the first level of rays will only use the first kind (possibly reflection) and the second level will only use the second kind (occlusion or shadow ray, for example) then the application can compute the stack size by something similar to:

\[
\text{rayGenStack} + \max(\text{closestHit1Stack, miss1Stack}) + \max(\text{closestHit2Stack, miss2Stack})
\]

This is guaranteed to be no larger than the default stack size computation which assumes that both call levels may be the larger of the two.
Chapter 39. Video Decode and Encode Operations

Vulkan implementations can expose video decode and encode engines, which are independent from the graphics and compute engines. Video decode and encode is performed by recording video operations and submitting them to video decode and encode queues. Vulkan provides core support for video decode and encode and can support a variety of video codecs through individual extensions built on the core video support.

The subsections below detail the fundamental components and operation of Vulkan video.

39.1. Technical Terminology and Semantics

39.1.1. Video Picture Resources

Video Picture Resources contain format information, can be multidimensional and may have associated metadata. The metadata can include implementation-private details required for the decode or encode operations and application managed color-space related information.

In Vulkan, a Video Picture Resource is represented by a VkImage. The VkImageView, representing the VkImage, is used with the decode operations as Output and Decoded Picture Buffer (Dpb), and with the encode operation as Input and Reconstructed Video Picture Resource.

39.1.2. Reference Picture

Video Reference Picture is a Video Picture Resource that can be used in the video decode or encode process to provide predictions of the values of samples in the subsequently decoded or encoded pictures.

39.1.3. Decoded Output Picture

The pixels resulting from the video decoding process are stored in a Decoded Output Picture, represented by a VkImageView. This can be shared with the Encoder Reconstructed or Decoder Dpb Video Picture Resources. It can also be used as an input for Video Encode, Graphics, Compute processing, or WSI presentation.

39.1.4. Input Picture to Encode

The primary source of input pixels for the video encoding process is the Input Picture to Encode, represented by a VkImageView. This can be shared with the Encoder Reconstructed or Decoder Dpb Video Picture Resources. It can be a direct target of Video Decode, Graphics, Compute processing, or WSI presentation.

39.1.5. Decoded Picture Buffer (Dpb)

Previously decoded pictures are used by video codecs to provide predictions of the values of samples in the subsequently decoded pictures. At the decoder, such Video Picture Resources are
stored in a **Decoded Picture Buffer (Dpb)** as an indexed set of **Reference Pictures**.

### 39.1.6. Reconstructed Pictures

An integral part of the video decoding pipeline is the reconstruction of pictures from the compressed stream. A similar stage exists in the video encoding pipeline as well. Such reconstructed pictures *may* be used as **Reference Pictures** for subsequently decoded or encoded pictures. The correct use of such **Reference Pictures** is driven by the video compression standard, the implementation, and the application-specific use cases.

This specification refers to the collection of the **Decoded Picture Buffer** and **Reconstructed Pictures** as **Decoded Picture Buffer (Dpb) Set**, or only, **Dpb**.

### 39.1.7. Decoded Picture Buffer (Dpb) Slot

**Decoded Picture Buffer (Dpb) Slot** represents a single or multi-layer indexed **Reference Picture**'s entry within the **Video Session's Dpb Set**. A valid **Dpb slot** index starts from zero and goes up to the maximum of **N - 1**, where **N** is the number of **Reference Picture** entries requested for a **Video Session**.

### 39.1.8. Reference Picture Metadata

The opaque **Dpb Slot** state managed by the implementation *may* contain **Reference Picture Metadata**, present when the **picture resource** associated with the **Dpb Slot** is used as a **reference picture** in one or more video decode or encode operations.

An implementation or application *may* have other **Picture Metadata** related to the **Video Picture Resource** or the **Dpb slot**, but such data is outside the scope of this specification.

> **Note:**
The video decode or encode implementation does not maintain internal references to the **Reference Pictures**, beyond the **Reference Picture Metadata**. It is the responsibility of the Vulkan Application to create, manage, and destroy, as well as to provide those **Video Picture Resources**, when required, during the decoding or encoding process.

### 39.1.9. Color Space Metadata

**Color Space Metadata** is the additional static or dynamic state associated with a **Video Picture Resource** specifying the color volume (the color primaries, white point, and luminance range) of the display that was used in mastering the video content. The use of **Color Space Metadata** is outside the scope of the current version of the video core specification.

### 39.2. Introduction

This chapter discusses extensions supporting Video Decode or Encode operations. Video Decode and Encode operations are supported by queues with an advertised queue capability of **VK_QUEUE_VIDEO_DECODE_BIT_KHR** and **VK_QUEUE_VIDEO_ENCODE_BIT_KHR**, respectively. Video Decode or
Encode queue operation support allows for Vulkan applications to cooperate closely with other graphics or compute operations seamlessly and efficiently, therefore improving the overall application performance.

### 39.2.1. Video Decode Queue

VK_KHR_video_decode_queue adds a video decode queue type bit `VK_QUEUE_VIDEO_DECODE_BIT_KHR` to `VkQueueFlagBits`. As in the case of other queue types, an application **must** use `vkGetPhysicalDeviceQueueFamilyProperties` to query whether the physical device has support for the Video Decode Queue. When the implementation reports the `VK_QUEUE_VIDEO_DECODE_BIT_KHR` bit for a queue family, it advertises general support for Vulkan queue operations described in Devices and Queues.

### 39.2.2. Video Encode Queue

VK_KHR_video_encode_queue adds a video encode queue type bit `VK_QUEUE_VIDEO_ENCODE_BIT_KHR` to `VkQueueFlagBits`. As in the case of other queue types, an application **must** use `vkGetPhysicalDeviceQueueFamilyProperties` to query whether the physical device has support for the Video Encode Queue. When the implementation reports the `VK_QUEUE_VIDEO_ENCODE_BIT_KHR` bit for a queue family, it advertises general support for Vulkan queue operations described in Devices and Queues.

The rest of the chapter focuses, specifically, on Video Decode and Encode queue operations.

### 39.2.3. Video Session

Before performing any video decoding or encoding operations, the application **must** create a Video Session instance, of type `VkVideoSessionKHR`. A Video Session instance is an immutable object and supports a single compression standard (for example, H.264, H.265, VP9, AV1, etc.). The implementation uses the `VkVideoSessionKHR` object to maintain the video state for the video decode or video encode operation. A Video Session instance is created specifically:

- For a particular video compression standard;
- For video decoding or video encoding;
- With maximum supported decoded or encoded picture width/height;
- With the maximum number of supported Dpb or Reconstructed Pictures slots that can be allocated;
- With the maximum number of Reference Pictures that **can** be used simultaneously for video decode or encode operations;
- Codec color and features profile;
- Color Space format description (not supported with this version of the specification);

`VkVideoSessionKHR` represents a single video decode or encode stream. For each concurrently used stream, a separate instance of `VkVideoSessionKHR` is required. After the application has finished with the processing of a stream, it can reuse the Video Session instance for another, provided that the configuration parameters between the two usages are compatible (as determined by the video...
compression standard in use). Once the VkVideoSessionKHR instance has been created, the video compression standard and profiles, Input / Output / Dpb formats, and the settings like the maximum extent cannot be changed.

The values of the following VkVideoSessionKHR parameters can be updated each frame, subject to the restrictions imposed on parameter updates by the video compression standard in use:

- decoded or encoded picture size
- number of active Dpb or Reconstructed Picture slots
- number of Reference Pictures in use,
- color space and color space metadata
- color space metadata.

The updated parameters must not exceed the maximum limits specified when creating the VkVideoSessionKHR instance.

39.2.4. Video Session Device Memory Heaps

After creating a Video Session instance, and before the object can be used for any of the decode or encode operations, the application must allocate and bind device memory resources to the Video Session object. An implementation may require one or more device memory heaps of different memory types, as reported by the vkGetVideoSessionMemoryRequirementsKHR function, to be bound with the vkBindVideoSessionMemoryKHR function to the Video Session. For more information about the Video Session Device Memory, please refer to the Binding the Session Object Device Memory section, below.

39.2.5. Video Session Parameters

A lot of codec standards require parameters that are in use for the entire video stream. For example, H.264/AVC and HEVC standards require sequence and picture parameter sets (SPS and PPS) that apply to multiple Video Decode and Encode frames, layers, and sub-layers. Vulkan Video uses Video Session Parameters objects to store such standard parameters. The application creates one or more Video Session Parameters Objects against a Video Session, with a set of common Video Parameters that are required for the processing of the video content. During the object creation, the implementation stores the parameters to the created instance. During command buffer recording, it is the responsibility of the application to provide the Video Session Parameters object containing the parameters that are necessary for the processing the portion of the stream under consideration.

39.2.6. Video Picture Subresources

For Video Picture Resources, an application has the option to use single or multi-layer images for image views. The layer to be used during decode or encode operations can be specified when the image view is being created with the VkImageSubresourceRange::baseArrayLayer parameter, and/or within the resource binding operations in command buffer by using the VkVideoPictureResourceKHR::baseArrayLayer parameter.
Both Video Decode and Encode operations only work with a single layer at the time.

The Image views representing the Input / Output / Dpb Video Picture Resources could have been created with sizes bigger than the coded size that is used with Video Decode and Encode operations. This allows for the same Video Picture Resources to be reused when there is a change in the input video content resolution. The effective coded size of the Video Picture Resources used for Video Decode and Encode operations is provided with VkVideoPictureResourceKHR::extent parameter of each resource in use.

Many codec standards require the coded and Video Picture Resources' sizes to match.

Video Session Dpb and Reconstructed Video Picture Resources

The video compression standard chosen may require the use of Reference Pictures. In Vulkan Video, like any other Video Picture Resources, the Reference Pictures are represented with Image Views.

When an application requires Reference Picture Resources, it creates and then associates image views, representing these resources, with Video Session Dpb or Reconstructed slots while recording the command buffer.

Decoded output pictures may be used as reference pictures in future video decode operations. The same pictures may be used in texture sampling operations or in the (WSI) presentation pipeline. Representing the Dpb's Video Picture Resources by image views makes it possible to accommodate all these use cases in a "zero-copy" fashion. Also, it provides more fine-grained control of the application over the efficient usage of the Dpb and Reconstructed Device Memory Resources.

Video Session Dpb and Reconstructed Slot Resource Management

Before Video Picture Resources can be used as Reference Picture Resources, Video Session Dpb or Reconstructed Slots must be associated with those resources.

The application allocates a Dpb or Reconstructed Slot and associates it with a Video Picture Resource and then sets up the resource as a target of decode or encode operation. After successfully decoding or encoding a picture with the targeted Dpb or Reconstructed Slot, in addition to the Reference Picture pixel data, the implementation may generate an opaque Reference Picture Metadata for that video session Slot and its associated Video Picture Resource.

Subsequently, one or more Dpb or Reconstructed video session Slots, along with their associated Video Picture Resources, can be used as Reference Picture's source for the video decode or encode operations.

If Reference Pictures were to be required for decoding or encoding of the video bitstream, the VkVideoSessionCreateInfoKHR::maxReferencePicturesSlotsCount must be set to a value bigger than "zero" when the instance of the Video Session object is created.
Up to \texttt{VkVideoSessionCreateInfoKHR::maxReferencePicturesSlotsCount} slots \textbf{can} be activated with \textit{Video Picture Resources} for a video session and up to \texttt{VkVideoSessionCreateInfoKHR::maxReferencePicturesActiveCount} active slots \textbf{can} be used as Dpb or Reconstructed \textit{Reference Pictures} within a single decode or encode operation.

When the implementation is associating \textit{Reference Picture Metadata} with the \textit{Video Picture Resources} themselves, such data \textbf{must} be independent of the \textit{Video Session} to allow for those \textit{Video Picture Resources} to be shared with other \textit{Video Session} instances. All of the \textit{Video Session}-dependent \textit{Reference Picture Metadata} \textbf{must} only be associated with the \textit{Video Session Dpb} or \textit{Reconstructed Slots}.

The application with the help of the implementation is responsible for managing the individual \textit{Dpb}, or \textit{Reconstructed Slots} that belong to a single \textit{Video Session Dpb set}:

- The application maintains the Slot allocation and per-slot \textit{Reference Picture Resources};
- Implementation maintains global and per-slot opaque \textit{Reference Picture Metadata};

The application also manages the mapping between the codec-specific picture IDs and \textit{Dpb Slots}.

When a Video Picture is decoded and is set as a \textit{Reference Picture} against a \textit{Video Session Dpb slot}, or is encoded and a Reconstructed Video Picture Resource is associated with a \textit{Video Session Dpb Slot} then:

- The \textit{Video Picture Resource} associated with the Slot is filled with the decoded or reconstructed pixel data;
- The implementation generates the \textit{Dpb Slot’s Reference Picture Metadata};

When a \textit{Dpb’s Slot} is deactivated, or a different \textit{Video Picture Resource} is used with the Slot, or the content of the \textit{Video Picture Resource} is modified, the \textit{Reference Picture Metadata} associated with the \textit{Dpb slot} gets invalidated by the implementation. Subsequent attempts to use such, invalidated, \textit{Dpb Slot} as a \textit{Reference} source would produce undefined results.

\textbf{Video Session Dpb Slot subresources}

\textit{Dpb Reference Picture’s} coded width and height \textbf{can} change, dynamically, via \texttt{VkVideoPictureResourceKHR::extent}, and the picture parameters from the codec-specific extensions. When a \textit{Dpb Slot} is activated as a \textit{Reference Picture} and a decode or encode operation is performed against that slot, the coded extent \textbf{can} be recorded by the implementation to the corresponding \textit{Dpb slot’s metadata state}. Subsequently, when the \textit{Reference Pictures} are used with the decoded \textit{Output} or encoded \textit{Input Picture}, their coded extent can differ. Decoding or encoding pictures, using picture sizes, different from the previously produced \textit{Reference Pictures} \textbf{should} be used with care, not to conflict with the codec standard and the implementation’s support for that. It is the responsibility of the application to ensure that valid \textit{Dpb Set} of \textit{Reference Pictures} are in use, according to the codec standard.

In addition, the \textit{Video Picture Resources} extent \textbf{cannot} exceed the \texttt{VkVideoSessionCreateInfoKHR::maxCodedExtent}.

2000 | Chapter 39. Video Decode and Encode Operations
**Note:**
Coding Standards such as VP9 and AV1 allow for images with different sizes to be used as Reference Pictures. Others, like H.264 and H.265, do not support Reference Pictures with different sizes. Using Reference Pictures with incompatible sizes with such standards would render undefined results.

**The application is in control of the allocation and use of the system resource**

In Vulkan Video, the application has complete control over how and when system resources are used. The Vulkan Video framework provides the following tools to ensure that device and host memory resources are used in an optimal way:

- The video application can allocate or destroy the number of allocated Output or Input Pictures, and can grow, or shrink the Dpb set of Reference Pictures, dynamically, based on the changing video content requirement.

- Reference Pictures can be shared with the decoded Output or encoded Input pictures.

- The application can use sparse memory for the images, representing Video Picture Resources. The use of sparse memory would allow the application to remove the Device Memory backing of the image resources when the Dpb Slot is not in active use. Furthermore, if the sparse residency feature is supported by the implementation (see Sparse Resources), then images can be, partially, bound with the resource memory. This feature is particularly important when using video content with a significant change of decoded or encoded resolution.

- If the implementation supports image arrays, and sparse memory resources, then the application can remove the Device Memory backing of image array layers that are not used by any Dpb Slots.

**Using Dpb and Reconstructed Slot’s Associated Resources**

Before a Dpb Slot is to become Valid for use with a Reference Picture, it requires memory resources to be bound to it.

Some of the memory resources required for the Dpb Slot, are opaquely managed by the implementation and, internally, allocated from the Session’s Device Memory Heaps. The application provides the image resources of one or more Reference Pictures, in the VkVideoBeginCodingInfoKHR::pReferenceEntries as part of the vkCmdBeginVideoCoding command.

If a Dpb slot was already used with an image view, and a new image view or a NULL handle is used with that Slot, then the Dpb Slot’s state will be invalidated by the implementation. If a Dpb Slot were to be reused with the same image view, the state of the Slot would not change.

**Video Session Activating Dpb Slot as a Reference**

Before a Dpb Slot is to be used for a Reference Pictures index, it must be activated. The activation of a Dpb Slot is done within the vkCmdDecodeVideoKHR command’s VkVideoDecodeInfoKHR::dpbSlotIndex field for the decode operations, and within the vkCmdEncodeVideoKHR command’s VkVideoEncodeInfoKHR::dpbSlotIndex field for the encode operations.

While activating a Slot for Dpb, it must already have an associated image view, within the
 VkVideoBeginCodingInfoKHR::pReferenceEntries in the vkCmdBeginVideoCodingKHR command and Device Memory backing of the image resources must be resident.

When a Dpb Slot were to be activated, the VkVideoDecodeInfoKHR::dpbSlotIndex for decode, or VkVideoDecodeInfoKHR::dpbSlotIndex for encode, must be set to the application's allocated Dpb Slot's index. When activating a Dpb Slot, the application will perform a decode or encode operation against its Slot's index in order to enable its state as a Valid Picture Reference. If a Dpb Slot is activated, but a decode or encode operation is not performed against that Slot's index, or the decode or encode operation was unsuccessful, then the Dpb Slot would remain in the Invalid Picture Reference state (see below the Dpb Slot States).

By just providing a Video Picture Resources for a Dpb Slot within the VkVideoBeginCodingInfoKHR::pReferenceEntries, and without successfully performing a decode or encode operation against that Slot, the Dpb Slot's state cannot be changed to Valid Picture Reference. If the Dpb slots were already in Valid Picture Reference, and there is no Video Picture Resources associated with the Dpb Slot for a decode or encode operation, the state Dpb Slot would not change. However, if an application is referring to a valid Dpb Slot in its current decode or encode operations, then a valid image view must be provided for that Slot within VkVideoPictureResourceKHR::imageViewBinding for that decode or encode operation.

Video Session Invalidating Dpb Slot's Reference State

When a Dpb Slot is invalidated, its state is set to Invalid Picture Reference. Using a Dpb Slot as a Reference Picture index for video decode or encode operations while the Slot is in Invalid Picture Reference state would render undefined results.

Video Session Dpb Slot States

To help understand the valid use of the Video Session Dpb and its resource management, this section aims to explain the different states and correct usage of Dpb Slots.

There are four (4) states that a Dpb slot could be in:

- Picture Reference Unused;
- Invalid Picture Reference;
- Updating Picture Reference;
- Valid Picture Reference;

The different states are outlined within the Dpb Slot States and Dpb Slot States Flow Diagram below.

All Dpb Slot management operations are performed via the VkVideoDecodeInfoKHR::dpbSlotIndex or VkVideoEncodeInfoKHR::dpbSlotIndex field.

All Dpb resource binding, invalidating, and activating Slot management operations are performed, by the implementation, before the decoding or encoding commands, based on the VkVideoDecodeInfoKHR::dpbSlotIndex or VkVideoEncodeInfoKHR::dpbSlotIndex field and the entries from the VkVideoBeginCodingInfoKHR::pReferenceEntries. The application cannot move a Dpb Slot from a Picture Reference Unused to Updating Picture Reference state, implicitly, within a decode or encode command operation. Such a Dpb Slot must first be transitioned to an Invalid...
**Picture Reference** state using `VkVideoDecodeInfoKHR::dpbSlotIndex` or `VkVideoEncodeInfoKHR::dpbSlotIndex`, as part of a decode command. For more details, see Video Picture Decode Modes.

When using **sparse memory resources**, it would be acceptable and valid behavior for the application to unbind the memory while the Dpb Slot is any of the Dpb Slot states, provided the command buffers, in a **pending state**, do not reference any such Video Picture Resources.

Accessing **unbound regions** of the sparse memory resources by the decoder or encoder, regardless if those are used as **Output, Input, Dpb** or **Reconstructed Video Picture Resources**, would render undefined results. The `VkPhysicalDeviceSparseProperties::residencyNonResidentStrict` property reported by the implementation does not offer guarantees on the behavior of decode or encode operations when it comes to accessing unbound regions. However, both reads and writes are still considered safe and will not affect other resources or populated regions of the image.

**Table 50. Video Session Dpb Slot States**

<table>
<thead>
<tr>
<th>Dpb Slot State</th>
<th>Moving to Dpb Slot State</th>
<th>Exiting Dpb Slot State</th>
<th>Retain Video Picture Resource Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Picture Reference</strong></td>
<td>* Bind Device Memory; * Reset decode or encode state; * Invalidate, delete or unbind memory of a Picture Reference associated with Reference Dpb Slot</td>
<td>* Activate Reference Dpb Slot → <strong>Invalid Picture Reference</strong></td>
<td><strong>Application Controlled</strong></td>
</tr>
<tr>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid Picture Reference</td>
<td>* Activate Reference Dpb Slot; * Unsuccessful video decode or encode operation;</td>
<td>* Start decode or encode operation with an active Reference Dpb Slot target → <strong>Updating Picture Reference</strong>; * Updating a Picture Resource outside the decoder or encoder or deleting or removing the memory binding(sparse) → <strong>Picture Reference Unused</strong>;</td>
<td><strong>Application Controlled</strong></td>
</tr>
<tr>
<td>Updating Picture Reference</td>
<td>Start decode or encode operation with an active Reference Dpb Slot target;</td>
<td>Decode or encode operation with an active Reference Dpb Slot target Completed Successfully → <strong>Valid Picture Reference</strong>; Unsuccessful video decode or encode operation → <strong>Invalid Picture Reference</strong></td>
<td>Yes</td>
</tr>
<tr>
<td>Valid Picture Reference</td>
<td>Video decode or encode operation with an active Reference Dpb Slot target Completed Successfully;</td>
<td>* Replace Reference Dpb slot → <strong>Invalid Picture Reference</strong>; * Invalidate, delete or unbind memory of a Picture Reference of the Reference Dpb Slot → <strong>Picture Reference Unused</strong>;</td>
<td>Yes</td>
</tr>
</tbody>
</table>
39.3. Video Physical Device Capabilities

39.3.1. Supported Video Codec Operations Enumeration

The structure `VkVideoQueueFamilyProperties2KHR` may be chained to `VkQueueFamilyProperties2` when calling `vkGetPhysicalDeviceQueueFamilyProperties2` to retrieve the video codec operations supported for the physical device queue family index.

The `VkVideoQueueFamilyProperties2KHR` structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoQueueFamilyProperties2KHR {
    VkStructureType sType;
    void* pNext;
    VkVideoCodecOperationFlagsKHR videoCodecOperations;
} VkVideoQueueFamilyProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `videoCodecOperations` is the supported video codec operation(s) as defined in `VkVideoCodecOperationFlagBitsKHR`.
Valid Usage (Implicit)

- **VUID-VkVideoQueueFamilyProperties2KHR-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_VIDEO_QUEUE_FAMILY_PROPERTIES_2_KHR`

- **VUID-VkVideoQueueFamilyProperties2KHR-videoCodecOperations-parameter**
  
  `videoCodecOperations` **must** be a valid combination of `VkVideoCodecOperationFlagBitsKHR` values

- **VUID-VkVideoQueueFamilyProperties2KHR-videoCodecOperations-requiredBitmask**
  
  `videoCodecOperations` **must** not be 0

The codec operations are defined with the `VkVideoCodecOperationFlagBitsKHR` enum:

```c
// Provided by VK_KHR_video_queue
typedef enum VkVideoCodecOperationFlagBitsKHR {
    VK_VIDEO_CODEC_OPERATION_INVALID_BIT_KHR = 0,
    #ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_encode_h264
    VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_EXT = 0x00010000,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h264
    VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_EXT = 0x00000001,
    #endif
    #ifdef VK_ENABLE_BETA_EXTENSIONS
    // Provided by VK_EXT_video_decode_h265
    VK_VIDEO_CODEC_OPERATION_DECODE_H265_BIT_EXT = 0x00000002,
    #endif
} VkVideoCodecOperationFlagBitsKHR;
```

Each decode or encode codec-specific extension extends this enumeration with the appropriate bit corresponding to the extension's codec operation:

- `VK_VIDEO_CODEC_OPERATION_INVALID_BIT_KHR` - No video operations are supported for this queue family.
- `VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_EXT` - H.264 video encode operations are supported by this queue family.
- `VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_EXT` - H.264 video decode operations are supported by this queue family.
- `VK_VIDEO_CODEC_OPERATION_DECODE_H265_BIT_EXT` - H.265 video decode operations are supported by this queue family.

```c
// Provided by VK_KHR_video_queue
typedef VkFlags VkVideoCodecOperationFlagsKHR;
```
 VkVideoCodecOperationFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoCodecOperationFlagBitsKHR.

39.3.2. Video Profiles

A video profile is defined by VkVideoProfileKHR structure as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoProfileKHR {
    VkStructureType                     sType;
    void*                               pNext;
    VkVideoCodecOperationFlagBitsKHR    videoCodecOperation;
    VkVideoChromaSubsamplingFlagsKHR    chromaSubsampling;
    VkVideoComponentBitDepthFlagsKHR    lumaBitDepth;
    VkVideoComponentBitDepthFlagsKHR    chromaBitDepth;
} VkVideoProfileKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **videoCodecOperation** is the specific video codec operation defined in VkVideoCodecOperationFlagBitsKHR.
- **chromaSubsampling** is the video chroma subsampling information defined in VkVideoChromaSubsamplingFlagBitsKHR.
- **lumaBitDepth** is the video luma bit depth information defined in VkVideoComponentBitDepthFlagBitsKHR.
- **chromaBitDepth** is the video chroma bit depth information defined in VkVideoComponentBitDepthFlagBitsKHR.
Valid Usage (Implicit)

- **VUID-VkVideoProfileKHR-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_VIDEO_PROFILE_KHR`.

- **VUID-VkVideoProfileKHR-videoCodecOperation-parameter**
  
  `videoCodecOperation` **must** be a valid `VkVideoCodecOperationFlagBitsKHR` value.

- **VUID-VkVideoProfileKHR-chromaSubsampling-parameter**
  
  `chromaSubsampling` **must** be a valid combination of `VkVideoChromaSubsamplingFlagBitsKHR` values.

- **VUID-VkVideoProfileKHR-chromaSubsampling-requiredbitmask**
  
  `chromaSubsampling` **must** not be 0.

- **VUID-VkVideoProfileKHR-chromaBitDepth-parameter**
  
  `chromaBitDepth` **must** be a valid combination of `VkVideoComponentBitDepthFlagBitsKHR` values.

- **VUID-VkVideoProfileKHR-chromaBitDepth-requiredbitmask**
  
  `chromaBitDepth` **must** not be 0.

The video format chroma subsampling is defined with the following enums:

```c
// Provided by VK_KHR_video_queue
typedef enum VkVideoChromaSubsamplingFlagBitsKHR {
    VK_VIDEO_CHROMA_SUBSAMPLING_INVALID_BIT_KHR = 0,
    VK_VIDEO_CHROMA_SUBSAMPLING_MONOCHROME_BIT_KHR = 0x00000001,
    VK_VIDEO_CHROMA_SUBSAMPLING_420_BIT_KHR = 0x00000002,
    VK_VIDEO_CHROMA_SUBSAMPLING_422_BIT_KHR = 0x00000004,
    VK_VIDEO_CHROMA_SUBSAMPLING_444_BIT_KHR = 0x00000008,
} VkVideoChromaSubsamplingFlagBitsKHR;
```

- **VK_VIDEO_CHROMA_SUBSAMPLING_MONOCHROME_BIT_KHR** - the format is monochrome.

- **VK_VIDEO_CHROMA_SUBSAMPLING_420_BIT_KHR** - the format is 4:2:0 chroma subsampled. The two chroma components are each subsampled at a factor of 2 both horizontally and vertically.

- **VK_VIDEO_CHROMA_SUBSAMPLING_422_BIT_KHR** - the format is 4:2:2 chroma subsampled. The two chroma components are sampled at half the sample rate of luma. The horizontal chroma resolution is halved.

- **VK_VIDEO_CHROMA_SUBSAMPLING_444_BIT_KHR** - the format is 4:4:4 chroma sampled. Each of the three YCbCr components have the same sample rate, thus there is no chroma subsampling.
VkVideoChromaSubsamplingFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoChromaSubsamplingFlagBitsKHR.

The video format component bit depth is defined with the following enums:

- **VK_VIDEO_COMPONENT_BIT_DEPTH_8_BIT_KHR** - the format component bit depth is 8 bits.
- **VK_VIDEO_COMPONENT_BIT_DEPTH_10_BIT_KHR** - the format component bit depth is 10 bits.
- **VK_VIDEO_COMPONENT_BIT_DEPTH_12_BIT_KHR** - the format component bit depth is 12 bits.

A video profile is provided when querying capabilities or image formats for video using vkGetPhysicalDeviceVideoCapabilitiesKHR or vkGetPhysicalDeviceVideoFormatPropertiesKHR, respectively. A video profile is also provided when creating resources (images, video sessions, etc.) used by video queues. Each instance of VkVideoProfileKHR must chain a codec-operation specific video profile extension structure, corresponding to the codec-operation specified in VkVideoProfileKHR::videoCodecOperation. Additional information is provided in each codec-operation-specific video extension.

### 39.3.3. Supported Video Decode or Encode Capabilities

To query video decode or encode capabilities for a specific codec, call:

```c
VkResult vkGetPhysicalDeviceVideoCapabilitiesKHR(
    VkPhysicalDevice physicalDevice,
    const VkVideoProfileKHR* pVideoProfile,
    VkVideoCapabilitiesKHR* pCapabilities);
```

- **physicalDevice** is the physical device whose video decode or encode capabilities will be queried.
• `pVideoProfile` is a pointer to an instance of the `VkVideoProfileKHR` structure with a chained codec-operation specific video profile structure.

• `pCapabilities` is a pointer to an instance of the `VkVideoCapabilitiesKHR` structure in which the capabilities are returned.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceVideoCapabilitiesKHR-physicalDevice-parameter
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceVideoCapabilitiesKHR-pVideoProfile-parameter
  - `pVideoProfile` must be a valid pointer to a valid `VkVideoProfileKHR` structure

- VUID-vkGetPhysicalDeviceVideoCapabilitiesKHR-pCapabilities-parameter
  - `pCapabilities` must be a valid pointer to a `VkVideoCapabilitiesKHR` structure

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_EXTENSION_NOT_PRESENT`
- `VK_ERROR_INITIALIZATION_FAILED`
- `VK_ERROR_FEATURE_NOT_PRESENT`
- `VK_ERROR_FORMAT_NOT_SUPPORTED`

If `pVideoProfile` and chained codec-operation specific profile is not supported, `VK_ERROR_FORMAT_NOT_SUPPORTED` is returned.

Otherwise, the implementation will fill `pCapabilities` with capabilities associated with this video profile.

The `VkVideoCapabilitiesKHR` structure is defined as:
typedef struct VkVideoCapabilitiesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkVideoCapabilitiesFlagsKHR capabilityFlags;
    VkDeviceSize       minBitstreamBufferOffsetAlignment;
    VkDeviceSize       minBitstreamBufferSizeAlignment;
    VkExtent2D         videoPictureExtentGranularity;
    VkExtent2D         minExtent;
    VkExtent2D         maxExtent;
    uint32_t           maxReferencePicturesSlotsCount;
    uint32_t           maxReferencePicturesActiveCount;
} VkVideoCapabilitiesKHR;

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **capabilityFlags** is the capability flags.
- **minBitstreamBufferOffsetAlignment** is the minimum alignment for the input or output bitstream buffer offset.
- **minBitstreamBufferSizeAlignment** is the minimum alignment for the input or output bitstream buffer size
- **videoPictureExtentGranularity** is the minimum size alignment of the extent with the required padding for the decoded or encoded video images.
- **minExtent** is the minimum width and height of the decoded or encoded video.
- **maxExtent** is the maximum width and height of the decoded or encoded video.
- **maxReferencePicturesSlotsCount** - the maximum number of Dpb slots supported by the implementation for a single video session instance.
- **maxReferencePicturesActiveCount** - is the maximum slots that can be used as Reference Pictures with a single decode or encode operation.

**Valid Usage (Implicit)**

- VUID-VkVideoCapabilitiesKHR-sType-sType
  - **sType** must be **VK_STRUCTURE_TYPE_VIDEO_CAPABILITIES_KHR**
- VUID-VkVideoCapabilitiesKHR-pNext-pNext
  - Each **pNext** member of any structure (including this one) in the **pNext** chain must be either **NULL** or a pointer to a valid instance of **VkVideoDecodeH264CapabilitiesEXT**, **VkVideoDecodeH265CapabilitiesEXT**, or **VkVideoEncodeH264CapabilitiesEXT**
- VUID-VkVideoCapabilitiesKHR-sType-unique
  - The **sType** value of each struct in the **pNext** chain must be unique

The **VkVideoCapabilitiesKHR** flags are defined with the following enumeration:
// Provided by VK_KHR_video_queue
typedef enum VkVideoCapabilitiesFlagBitsKHR {
    VK_VIDEO_CAPABILITIES_PROTECTED_CONTENT_BIT_KHR = 0x00000001,
    VK_VIDEO_CAPABILITIES_SEPARATE_REFERENCE_IMAGES_BIT_KHR = 0x00000002,
} VkVideoCapabilitiesFlagBitsKHR;

- **VK_VIDEO_CAPABILITIES_PROTECTED_CONTENT_BIT_KHR** - the decode or encode session supports protected content.
- **VK_VIDEO_CAPABILITIES_SEPARATE_REFERENCE_IMAGES_BIT_KHR** - the DPB or Reconstructed Video Picture Resources for the video session may be created as a separate VkImage for each DPB picture. If not supported, the DPB must be created as single multi-layered image where each layer represents one of the DPB Video Picture Resources.

// Provided by VK_KHR_video_queue
typedef VkFlags VkVideoCapabilitiesFlagsKHR;

VkVideoCapabilitiesFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoCapabilitiesFlagBitsKHR.

### 39.3.4. Enumeration of Supported Video Output, Input and Dpb Formats

To enumerate the supported output, input and Dpb image formats for a specific codec operation and video profile, call:

// Provided by VK_KHR_video_queue
VkResult vkGetPhysicalDeviceVideoFormatPropertiesKHR(
    VkPhysicalDevice                            physicalDevice,
    const VkPhysicalDeviceVideoFormatInfoKHR*   pVideoFormatInfo,
    uint32_t*                                   pVideoFormatPropertyCount,
    VkVideoFormatPropertiesKHR*                 pVideoFormatProperties);

- **physicalDevice** is the physical device being queried.
- **pVideoFormatInfo** is a pointer to an instance of VkPhysicalDeviceVideoFormatInfoKHR, the input structure.
- **pVideoFormatPropertyCount** is a pointer to an integer related to the number of video format properties available or queried, as described below.
- **pVideoFormatProperties** is a pointer to an array of VkVideoFormatPropertiesKHR, where the implementation would populate the supported formats.

If **pVideoFormatProperties** is NULL, then the number of video format properties supported for the given **physicalDevice** is returned in **pVideoFormatPropertyCount**. Otherwise, **pVideoFormatPropertyCount** must point to a variable set by the user to the number of elements in the **pVideoFormatProperties** array, and on return the variable is overwritten with the number of values actually written to **pVideoFormatProperties**. If the value of **pVideoFormatPropertyCount** is less than the
number of video format properties supported, at most \( p_{\text{VideoFormatPropertyCount}} \) values will be written to \( p_{\text{VideoFormatProperties}} \), and \( \text{VK_INCOMPLETE} \) will be returned instead of \( \text{VK_SUCCESS} \), to indicate that not all the available values were returned.

### Valid Usage

- **VUID-vkGetPhysicalDeviceVideoFormatPropertiesKHR-imageUsage-04844**
  
  The `imageUsage` enum of \( \text{VkPhysicalDeviceVideoFormatInfoKHR} \) must contain at least one of the following video image usage bit(s): \( \text{VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR} \), \( \text{VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR} \), \( \text{VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR} \), or \( \text{VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR} \).

### Note:

For most use cases, only decode or encode related usage flags are going to be specified. For a use case such as transcode, if the image were to be shared between decode and encode session(s), then both decode and encode related usage flags can be set.

### Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceVideoFormatPropertiesKHR-physicalDevice-parameter**
  
  `physicalDevice` must be a valid \( \text{VkPhysicalDevice} \) handle

- **VUID-vkGetPhysicalDeviceVideoFormatPropertiesKHR-pVideoFormatInfo-parameter**
  
  `pVideoFormatInfo` must be a valid pointer to a valid \( \text{VkPhysicalDeviceVideoFormatInfoKHR} \) structure

- **VUID-vkGetPhysicalDeviceVideoFormatPropertiesKHR-pVideoFormatPropertyCount-parameter**
  
  `pVideoFormatPropertyCount` must be a valid pointer to a uint32_t value

- **VUID-vkGetPhysicalDeviceVideoFormatPropertiesKHR-pVideoFormatProperties-parameter**
  
  If \( p_{\text{VideoFormatProperties}} \) is not NULL, \( p_{\text{VideoFormatProperties}} \) must be a valid pointer to an array of \( p_{\text{VideoFormatPropertyCount}} \) \( \text{VkVideoFormatPropertiesKHR} \) structures

- **VUID-vkGetPhysicalDeviceVideoFormatPropertiesKHR-pVideoFormatPropertyCount-arraylength**
  
  If \( p_{\text{VideoFormatProperties}} \) is not NULL, the value referenced by \( p_{\text{VideoFormatPropertyCount}} \) must be greater than 0
Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_EXTENSION_NOT_PRESENT
- VK_ERROR_INITIALIZATION_FAILED
- VK_ERROR_FORMAT_NOT_SUPPORTED

The VkPhysicalDeviceVideoFormatInfoKHR input structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkPhysicalDeviceVideoFormatInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkImageUsageFlags imageUsage;
    const VkVideoProfilesKHR* pVideoProfiles;
} VkPhysicalDeviceVideoFormatInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `imageUsage` is the intended video image usage as specified in `VkImageUsageFlagBits`.
- `pVideoProfiles` is a pointer to an instance of `VkVideoProfilesKHR` providing the video profile(s) of video session(s) that will use the image. For most use cases, the image is used by a single video session and a single video profile is provided. For a use case such as transcode, where a decode session output image may be used as encode input for one or more encode sessions, multiple video profiles representing the video sessions that will share the image may be provided.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVideoFormatInfoKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VIDEO_FORMAT_INFO_KHR`
- VUID-VkPhysicalDeviceVideoFormatInfoKHR-pNext-pNext
  `pNext` must be `NULL`

The VkVideoProfilesKHR structure is defined as:
// Provided by VK_KHR_video_queue

typedef struct VkVideoProfilesKHR {
    VkStructureType             sType;
    void*                       pNext;
    uint32_t                    profileCount;
    const VkVideoProfileKHR*    pProfiles;
} VkVideoProfilesKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **profileCount** is an integer which holds the number of video profiles included in **pProfiles**.
- **pProfiles** is a pointer to an array of **VkVideoProfileKHR** structures. Each **VkVideoProfileKHR** structure must chain the corresponding codec-operation specific extension video profile structure.

### Valid Usage (Implicit)

- VUID-VkVideoProfilesKHR-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_VIDEO_PROFILES_KHR**

- VUID-VkVideoProfilesKHR-pProfiles-parameter
  
  **pProfiles** must be a valid pointer to a valid **VkVideoProfileKHR** structure

The **VkVideoFormatPropertiesKHR** output structure for **vkGetPhysicalDeviceVideoFormatPropertiesKHR** is defined as:

// Provided by VK_KHR_video_queue

typedef struct VkVideoFormatPropertiesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkFormat           format;
} VkVideoFormatPropertiesKHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **format** is one of the supported formats reported by the implementation.

If the **pVideoProfiles** provided in input structure **pVideoFormatInfo** are not supported, **VK_ERROR_FORMAT_NOT_SUPPORTED** is returned. If the implementation requires an opaque video decode or encode DPB, then when querying with the corresponding video decode or encode DPB image usage in **imageUsage**, only one image format is returned: **VK_FORMAT_UNDEFINED**.
Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_VIDEO_FORMAT_PROPERTIES_KHR`
- **pNext** must be `NULL`

Before creating an image, the application should obtain the supported image creation parameters by querying with `vkGetPhysicalDeviceFormatProperties2` or `vkGetPhysicalDeviceImageFormatProperties2` using one of the reported `pImageFormats` and adding `VkVideoProfilesKHR` to the `pNext` chain of `VkFormatProperties2`.

### 39.4. Video Session Objects

#### 39.4.1. Video Session

Video session objects are abstracted and represented by `VkVideoSessionKHR` handles:

```cpp
// Provided by VK_KHR_video_queue
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkVideoSessionKHR)
```

### Creating a Video Session

To create a video session object, call:

```cpp
// Provided by VK_KHR_video_queue
VkResult vkCreateVideoSessionKHR(
    VkDevice device,
    const VkVideoSessionCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkVideoSessionKHR* pVideoSession);
```

- **device** is the logical device that creates the decode or encode session object.
- **pCreateInfo** is the pointer to `VkVideoSessionCreateInfoKHR` structure containing parameters specifying the creation of the decode or encode session.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pVideoSession** is the pointer for the decode or encode video session object which will be created by this function when it returns `VK_SUCCESS`
Valid Usage (Implicit)

- VUID-vkCreateVideoSessionKHR-device-parameter
device must be a valid VkDevice handle

- VUID-vkCreateVideoSessionKHR-pCreateInfo-parameter
pCreateInfo must be a valid pointer to a valid VkVideoSessionCreateInfoKHR structure

- VUID-vkCreateVideoSessionKHR-pAllocator-parameter
If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkCreateVideoSessionKHR-pVideoSession-parameter
pVideoSession must be a valid pointer to a VkVideoSessionKHR handle

Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_INITIALIZATION_FAILED
- VK_ERROR_INCOMPATIBLE_DRIVER
- VK_ERROR_FEATURE_NOT_PRESENT

The VkVideoSessionCreateInfoKHR structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoSessionCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t queueFamilyIndex;
    VkVideoSessionCreateFlagsKHR flags;
    const VkVideoProfileKHR* pVideoProfile;
    VkFormat pictureFormat;
    VkExtent2D maxCodedExtent;
    VkFormat referencePicturesFormat;
    uint32_t maxReferencePicturesSlotsCount;
    uint32_t maxReferencePicturesActiveCount;
} VkVideoSessionCreateInfoKHR;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
• `queueFamilyIndex` is the queue family of the created video session.

• `flags` is the creation flags defined in `VkVideoSessionCreateFlagBitsKHR`.

• `pVideoProfile` is a pointer to an instance of `VkVideoProfileKHR`.

• `pictureFormat` is the format of the image views representing decoded Output or encoded Input pictures.

• `maxCodedExtent` is the maximum width and height of the coded pictures that this instance will be able to support.

• `referencePicturesFormat` is the format of the Dpb image views representing the Reference pictures.

• `maxReferencePicturesSlotsCount` is the maximum number of Dpb slots that can be activated with associated Video Picture Resources for the created video session.

• `maxReferencePicturesActiveCount` is the maximum number of active Dpb slots that can be used as Dpb or Reconstructed Reference Pictures within a single decode or encode operation for the created video session.
Valid Usage

- **VUID-VkVideoSessionCreateInfoKHR-pVideoProfile-04845**
  
  **pVideoProfile must** be a pointer to a valid `VkVideoProfileKHR` structure whose pNext chain must include a valid codec-specific profile structure.

- **VUID-VkVideoSessionCreateInfoKHR-maxReferencePicturesSlotsCount-04846**

  If **Reference Pictures** are required for use with the created video session, the `maxReferencePicturesSlotsCount` must be set to a value bigger than "zero".

- **VUID-VkVideoSessionCreateInfoKHR-maxReferencePicturesSlotsCount-04847**

  `maxReferencePicturesSlotsCount` cannot exceed the implementation reported `VkVideoCapabilitiesKHR::maxReferencePicturesSlotsCount`.

- **VUID-VkVideoSessionCreateInfoKHR-maxReferencePicturesActiveCount-04848**

  If **Reference Pictures** are required for use with the created video session, the `maxReferencePicturesActiveCount` must be set to a value bigger than "zero".

- **VUID-VkVideoSessionCreateInfoKHR-maxReferencePicturesActiveCount-04849**

  `maxReferencePicturesActiveCount` cannot exceed the implementation reported `VkVideoCapabilitiesKHR::maxReferencePicturesActiveCount`.

- **VUID-VkVideoSessionCreateInfoKHR-maxReferencePicturesActiveCount-04850**

  `maxReferencePicturesActiveCount` cannot exceed the `maxReferencePicturesSlotsCount`.

- **VUID-VkVideoSessionCreateInfoKHR-maxCodedExtent-04851**

  `maxCodedExtent` cannot be smaller than `VkVideoCapabilitiesKHR::minExtent` and bigger than `VkVideoCapabilitiesKHR::maxExtent`.

- **VUID-VkVideoSessionCreateInfoKHR-referencePicturesFormat-04852**

  **referencePicturesFormat must** be one of the supported formats in `VkVideoFormatPropertiesKHR::format` returned by the function `vkGetPhysicalDeviceVideoFormatPropertiesKHR` when the `VkPhysicalDeviceVideoFormatInfoKHR::imageUsage` contains `VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR` or `VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR` depending on the session codec operation.

- **VUID-VkVideoSessionCreateInfoKHR-pictureFormat-04853**

  **pictureFormat for decode output must** be one of the supported formats in `VkVideoFormatPropertiesKHR::format` returned by the function `vkGetPhysicalDeviceVideoFormatPropertiesKHR` when the `VkPhysicalDeviceVideoFormatInfoKHR::imageUsage` contains `VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR`.

- **VUID-VkVideoSessionCreateInfoKHR-pictureFormat-04854**

  **pictureFormat targeting encode operations must** be one of the supported formats in `VkVideoFormatPropertiesKHR::format` returned by the function `vkGetPhysicalDeviceVideoFormatPropertiesKHR` when the `VkPhysicalDeviceVideoFormatInfoKHR::imageUsage` contains `VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR`.
Valid Usage (Implicit)

- VUID-VkVideoSessionCreateInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_SESSION_CREATE_INFO_KHR

- VUID-VkVideoSessionCreateInfoKHR-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkVideoDecodeH264SessionCreateInfoEXT, VkVideoDecodeH265SessionCreateInfoEXT, or VkVideoEncodeH264SessionCreateInfoEXT

- VUID-VkVideoSessionCreateInfoKHR-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkVideoSessionCreateInfoKHR-flags-parameter
  flags must be a valid combination of VkVideoSessionCreateFlagBitsKHR values

- VUID-VkVideoSessionCreateInfoKHR-pVideoProfile-parameter
  pVideoProfile must be a valid pointer to a valid VkVideoProfileKHR structure

- VUID-VkVideoSessionCreateInfoKHR-pictureFormat-parameter
  pictureFormat must be a valid VkFormat value

- VUID-VkVideoSessionCreateInfoKHR-referencePicturesFormat-parameter
  referencePicturesFormat must be a valid VkFormat value

The decode or encode session creation flags defined with the following enums:

```c
// Provided by VK_KHR_video_queue
typedef enum VkVideoSessionCreateFlagBitsKHR {
    VK_VIDEO_SESSION_CREATE_DEFAULT_KHR = 0,
    VK_VIDEO_SESSION_CREATE_PROTECTED_CONTENT_BIT_KHR = 0x00000001,
} VkVideoSessionCreateFlagBitsKHR;
```

- **VK_VIDEO_SESSION_CREATE_PROTECTED_CONTENT_BIT_KHR** - create the video session for use with protected video content

```c
// Provided by VK_KHR_video_queue
typedef VkFlags VkVideoSessionCreateFlagsKHR;
```

VkVideoSessionCreateFlagBitsKHR is a bitmask type for setting a mask of zero or more VkVideoSessionCreateFlagBitsKHR.

39.4.2. Destroying a Video Session

To destroy a decode session object, call:
// Provided by VK_KHR_video_queue
void vkDestroyVideoSessionKHR(
    VkDevice                                    device,  // device is the device that was used for the creation of the video session.
    VkVideoSessionKHR                           videoSession,  // videoSession is the decode or encode video session to be destroyed.
    const VkAllocationCallbacks*                pAllocator);  // pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage (Implicit)

- VUID-vkDestroyVideoSessionKHR-device-parameter
  device must be a valid VkDevice handle

- VUID-vkDestroyVideoSessionKHR-videoSession-parameter
  videoSession must be a valid VkVideoSessionKHR handle

- VUID-vkDestroyVideoSessionKHR-pAllocator-parameter
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

- VUID-vkDestroyVideoSessionKHR-videoSession-parent
  videoSession must have been created, allocated, or retrieved from device

39.4.3. Video Session Memory Resource Management

Obtaining the Video Session Object Device Memory Requirements

To get memory requirements for a video session, call:

// Provided by VK_KHR_video_queue
VkResult vkGetVideoSessionMemoryRequirementsKHR(
    VkDevice                                    device,  // device is the logical device that owns the video session.
    VkVideoSessionKHR                           videoSession,  // videoSession is the video session to query.
    uint32_t*                                   pVideoSessionMemoryRequirementsCount,  // pVideoSessionMemoryRequirementsCount is the number of the memory requirements structures of type VkVideoGetMemoryPropertiesKHR that pVideoSessionMemoryRequirements points to.
    VkVideoGetMemoryPropertiesKHR*              pVideoSessionMemoryRequirements);  // pVideoSessionMemoryRequirements is the pointer to an instance of the VkVideoGetMemoryPropertiesKHR in which the memory heap requirements of the video session are returned - one entry for each memory heap type required by the implementation.
If pVideoSessionMemoryRequirements is NULL, then the number of memory heap types required for the video session is returned in pVideoSessionMemoryRequirementsCount. Otherwise, pVideoSessionMemoryRequirementsCount must point to a variable set by the user with the number of elements in the pVideoSessionMemoryRequirements array, and on return the variable is overwritten with the number of formats actually written to pVideoSessionMemoryRequirements. If pVideoSessionMemoryRequirementsCount is less than the number of memory heap types required for the video session then at most pVideoSessionMemoryRequirementsCount elements will be written to pVideoSessionMemoryRequirements, and VK_INCOMPLETE will be returned, instead of VK_SUCCESS, to indicate that not all required memory heap types were returned.

Valid Usage (Implicit)

- VUID-vkGetVideoSessionMemoryRequirementsKHR-device-parameter
device must be a valid VkDevice handle
- VUID-vkGetVideoSessionMemoryRequirementsKHR-videoSession-parameter
videoSession must be a valid VkVideoSessionKHR handle
- VUID-vkGetVideoSessionMemoryRequirementsKHR-pVideoSessionMemoryRequirementsCount-parameter
pVideoSessionMemoryRequirementsCount must be a valid pointer to a uint32_t value
- VUID-vkGetVideoSessionMemoryRequirementsKHR-pVideoSessionMemoryRequirements-parameter
If pVideoSessionMemoryRequirements is not NULL, pVideoSessionMemoryRequirements must be a valid pointer to an array of pVideoSessionMemoryRequirementsCount VkVideoGetMemoryPropertiesKHR structures
- VUID-vkGetVideoSessionMemoryRequirementsKHR-pVideoSessionMemoryRequirementsCount-arraylength
If pVideoSessionMemoryRequirements is not NULL, the value referenced by pVideoSessionMemoryRequirementsCount must be greater than 0
- VUID-vkGetVideoSessionMemoryRequirementsKHR-videoSession-parent
videoSession must have been created, allocated, or retrieved from device

Return Codes

Success

- VK_SUCCESS
- VK_INCOMPLETE

Failure

- VK_ERROR_INITIALIZATION_FAILED

The VkVideoGetMemoryPropertiesKHR structure is defined as:
typedef struct VkVideoGetMemoryPropertiesKHR {
    VkStructureType           sType;
    const void*               pNext;
    uint32_t                  memoryBindIndex;
    VkMemoryRequirements2*    pMemoryRequirements;
} VkVideoGetMemoryPropertiesKHR;

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `memoryBindIndex` is the memory bind index of the memory heap type described by the information returned in `pMemoryRequirements`.
- `pMemoryRequirements` returns the requested memory heap requirements entry of type `VkMemoryRequirements2` for heap with index `memoryBindIndex`.

Valid Usage (Implicit)

- VUID-VkVideoGetMemoryPropertiesKHR-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_VIDEO_GET_MEMORY_PROPERTIES_KHR`
- VUID-VkVideoGetMemoryPropertiesKHR-pNext-pNext
  `pNext` must be NULL
- VUID-VkVideoGetMemoryPropertiesKHR-pMemoryRequirements-parameter
  `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements2` structure

Binding the Session Object Device Memory

To attach memory to a video session object, call:

```c
// Provided by VK_KHR_video_queue
VkResult vkBindVideoSessionMemoryKHR(
    VkDevice                                    device,
    VkVideoSessionKHR                           videoSession,
    uint32_t                                    videoSessionBindMemoryCount,
    const VkVideoBindMemoryKHR*                 pVideoSessionBindMemories);
```

- `device` is the logical device that owns the video session’s memory.
- `videoSession` is the video session to be bound with device memory.
- `videoSessionBindMemoryCount` is the number of `pVideoSessionBindMemories` to be bound.
- `pVideoSessionBindMemories` is a pointer to the device memory heap entries of type `VkVideoBindMemoryKHR` which are to be bound to the video session.
Valid Usage (Implicit)

- **VUID-vkBindVideoSessionMemoryKHR-device-parameter**
  - `device` must be a valid `VkDevice` handle

- **VUID-vkBindVideoSessionMemoryKHR-videoSession-parameter**
  - `videoSession` must be a valid `VkVideoSessionKHR` handle

- **VUID-vkBindVideoSessionMemoryKHR-pVideoSessionBindMemories-parameter**
  - `pVideoSessionBindMemories` must be a valid pointer to an array of `VkVideoBindMemoryKHR` structures

- **VUID-vkBindVideoSessionMemoryKHR-videoSessionBindMemoryCount-arraylength**
  - `videoSessionBindMemoryCount` must be greater than 0

- **VUID-vkBindVideoSessionMemoryKHR-videoSession-parent**
  - `videoSession` must have been created, allocated, or retrieved from `device`

Return Codes

**Success**
- **VK_SUCCESS**

**Failure**
- **VK_ERROR_OUT_OF_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**
- **VK_ERROR_INITIALIZATION_FAILED**

The `VkVideoBindMemoryKHR` structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoBindMemoryKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t memoryBindIndex;
    VkDeviceMemory memory;
    VkDeviceSize memoryOffset;
    VkDeviceSize memorySize;
} VkVideoBindMemoryKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **memoryBindIndex** is the index of the device memory heap returned in `VkVideoGetMemoryPropertiesKHR::memoryBindIndex` from `vkGetVideoSessionMemoryRequirementsKHR`.
- **memory** is the allocated device memory to be bound to the video session’s heap with index
memoryBindIndex.

- memoryOffset is the start offset of the region of memory which is to be bound.
- memorySize is the size in bytes of the region of memory, starting from memoryOffset bytes, to be bound.

Valid Usage (Implicit)

- VUID-VkVideoBindMemoryKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_BIND_MEMORY_KHR
- VUID-VkVideoBindMemoryKHR-pNext-pNext
  pNext must be NULL
- VUID-VkVideoBindMemoryKHR-memory-parameter
  memory must be a valid VkDeviceMemory handle

39.4.4. Video Session Parameters

This specification supports several classes of preprocessed parameters stored in Video Session Parameters objects. The Video Session Parameters objects reduces the number of parameters being dispatched and then processed by the implementation while recording video command buffers.

39.4.5. Creating Video Session Parameters

Video session parameter objects are represented by VkVideoSessionParametersKHR handles:

```c
// Provided by VK_KHR_video_queue
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkVideoSessionParametersKHR)
```

To create a video session parameters object, call:

```c
// Provided by VK_KHR_video_queue
VkResult vkCreateVideoSessionParametersKHR(
    VkDevice                                    device,
    const VkVideoSessionParametersCreateInfoKHR* pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkVideoSessionParametersKHR*                pVideoSessionParameters);
```

- device is the logical device that was used for the creation of the video session object.
- pCreateInfo is a pointer to VkVideoSessionParametersCreateInfoKHR structure specifying the video session parameters.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.
- pVideoSessionParameters is the pointer for the video session parameters object returned by the function on VK_SUCCESS.
Valid Usage (Implicit)

- **VUID-vkCreateVideoSessionParametersKHR-device-parameter**
  
  *device* must be a valid VkDevice handle

- **VUID-vkCreateVideoSessionParametersKHR-pCreateInfo-parameter**
  
  *pCreateInfo* must be a valid pointer to a valid VkVideoSessionParametersCreateInfoKHR structure

- **VUID-vkCreateVideoSessionParametersKHR-pAllocator-parameter**
  
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid VkAllocationCallbacks structure

- **VUID-vkCreateVideoSessionParametersKHR-pVideoSessionParameters-parameter**
  
  *pVideoSessionParameters* must be a valid pointer to a VkVideoSessionParametersKHR handle

Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_TOO_MANY_OBJECTS

The **VkVideoSessionParametersCreateInfoKHR** structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoSessionParametersCreateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkVideoSessionParametersKHR videoSessionParametersTemplate;
    VkVideoSessionKHR videoSession;
} VkVideoSessionParametersCreateInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **videoSessionParametersTemplate** is an optional handle to an existing Video Session Parameters object. If this parameter represents a valid handle, then the underlying Video Session Parameters object will be used as a template for constructing the new video session parameters object. All of the template object's current parameters will be inherited by the new object in such a case. Optionally, some of the template's parameters can be updated or new parameters added to the newly constructed object via the extension-specific parameters.
• **videoSession** is the video session object against which the video session parameters object is going to be created.

### Valid Usage

- VUID-VkVideoSessionParametersCreateInfoKHR-videoSessionParametersTemplate-04855
  
  If **videoSessionParametersTemplate** represents a valid handle, it **must** have been created against **videoSession**.

### Valid Usage (Implicit)

- VUID-VkVideoSessionParametersCreateInfoKHR-sType-sType
  
  *sType** must be **VK_STRUCTURE_TYPE_VIDEO_SESSION_PARAMETERS_CREATE_INFO_KHR**

- VUID-VkVideoSessionParametersCreateInfoKHR-pNext-pNext
  
  Each **pNext** member of any structure (including this one) in the **pNext** chain **must** be either
  
  - NULL
  - or a pointer to a valid instance of
  
  - **VkVideoDecodeH264SessionParametersCreateInfoEXT**
  - **VkVideoDecodeH265SessionParametersCreateInfoEXT**
  - or
  
  - **VkVideoEncodeH264SessionParametersCreateInfoEXT**

- VUID-VkVideoSessionParametersCreateInfoKHR-sType-unique
  
  The **sType** value of each struct in the **pNext** chain **must** be unique

- VUID-VkVideoSessionParametersCreateInfoKHR-videoSessionParametersTemplate-parameter
  
  **videoSessionParametersTemplate** **must** be a valid **VkVideoSessionParametersKHR** handle

- VUID-VkVideoSessionParametersCreateInfoKHR-videoSession-parameter
  
  **videoSession** **must** be a valid **VkVideoSessionKHR** handle

- VUID-VkVideoSessionParametersCreateInfoKHR-videoSessionParametersTemplate-parent
  
  **videoSessionParametersTemplate** **must** have been created, allocated, or retrieved from **videoSession**

- VUID-VkVideoSessionParametersCreateInfoKHR-commonparent
  
  Both of **videoSession**, and **videoSessionParametersTemplate** **must** have been created, allocated, or retrieved from the same **VkDevice**

### 39.4.6. Updating the parameters of the Video Session Parameters object

To update, add, or remove video session parameters state, call:

```c
// Provided by VK_KHR_video_queue
VkResult vkUpdateVideoSessionParametersKHR(
    VkDevice                                    device,
    VkVideoSessionParametersKHR                 videoSessionParameters,
    const VkVideoSessionParametersUpdateInfoKHR* pUpdateInfo);
```

- **device** is the logical device that was used for the creation of the video session object.
• `videoSessionParameters` is the video session object that is going to be updated.

• `pUpdateInfo` is a pointer to `VkVideoSessionParametersUpdateInfoKHR` structure containing the session parameters update info.

### Valid Usage (Implicit)

- VUID-vkUpdateVideoSessionParametersKHR-device-parameter device must be a valid `VkDevice` handle

- VUID-vkUpdateVideoSessionParametersKHR-videoSessionParameters-parameter `videoSessionParameters` must be a valid `VkVideoSessionParametersKHR` handle

- VUID-vkUpdateVideoSessionParametersKHR-pUpdateInfo-parameter `pUpdateInfo` must be a valid pointer to a valid `VkVideoSessionParametersUpdateInfoKHR` structure

### Return Codes

**Success**

- `VK_SUCCESS`

**Failure**

- `VK_ERROR_INITIALIZATION_FAILED`
- `VK_ERROR_TOO_MANY_OBJECTS`

The `VkVideoSessionParametersUpdateInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoSessionParametersUpdateInfoKHR {
    VkStructureType sType;
    const void* pNext;
    uint32_t updateSequenceCount;
} VkVideoSessionParametersUpdateInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `updateSequenceCount` is the sequence number of the object update with parameters, starting from "1" and incrementing the value by one with each subsequent update.
Valid Usage (Implicit)

- VUID-VkVideoSessionParametersUpdateInfoKHR-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_VIDEO_SESSION_PARAMETERS_UPDATE_INFO_KHR

- VUID-VkVideoSessionParametersUpdateInfoKHR-pNext-pNext
  
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkVideoDecodeH264SessionParametersAddInfoEXT, VkVideoDecodeH265SessionParametersAddInfoEXT, or VkVideoEncodeH264SessionParametersAddInfoEXT

- VUID-VkVideoSessionParametersUpdateInfoKHR-sType-unique
  
  The sType value of each struct in the pNext chain must be unique

39.4.7. Destroying Video Session Parameters

To destroy a video session object, call:

```c
// Provided by VK_KHR_video_queue
void vkDestroyVideoSessionParametersKHR(
    VkDevice device,
    VkVideoSessionParametersKHR videoSessionParameters,
    const VkAllocationCallbacks* pAllocator);
```

- **device** is the device the video session was created with.
- **videoSessionParameters** is the video session parameters object to be destroyed.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage (Implicit)

- VUID-vkDestroyVideoSessionParametersKHR-device-parameter
  
  device must be a valid VkDevice handle

- VUID-vkDestroyVideoSessionParametersKHR-videoSessionParameters-parameter
  
  videoSessionParameters must be a valid VkVideoSessionParametersKHR handle

- VUID-vkDestroyVideoSessionParametersKHR-pAllocator-parameter
  
  If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

39.4.8. Video Encode and Decode commands

To start video decode or encode operations, call:
void vkCmdBeginVideoCodingKHR(
  VkCommandBuffer                             commandBuffer,
  const VkVideoBeginCodingInfoKHR*            pBeginInfo);

- `commandBuffer` is the command buffer to be used when recording commands for the video decode and encode operations.
- `pBeginInfo` is the pointer to an instance of `VkVideoBeginCodingInfoKHR`.

### Valid Usage (Implicit)

- VUID-vkCmdBeginVideoCodingKHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdBeginVideoCodingKHR-pBeginInfo-parameter
  `pBeginInfo` must be a valid pointer to a valid `VkVideoBeginCodingInfoKHR` structure

- VUID-vkCmdBeginVideoCodingKHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdBeginVideoCodingKHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support decode, or encode operations

- VUID-vkCmdBeginVideoCodingKHR-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdBeginVideoCodingKHR-bufferlevel
  `commandBuffer` must be a primary `VkCommandBuffer`

### Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Decode, Encode</td>
<td></td>
</tr>
</tbody>
</table>

The `VkVideoBeginCodingInfoKHR` structure is defined as:
typedef struct VkVideoBeginCodingInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkVideoBeginCodingFlagsKHR flags;
    VkVideoCodingQualityPresetFlagsKHR codecQualityPreset;
    VkVideoSessionKHR videoSession;
    VkVideoSessionParametersKHR videoSessionParameters;
    uint32_t referenceSlotCount;
    const VkVideoReferenceSlotKHR* pReferenceSlots;
} VkVideoBeginCodingInfoKHR;

• `sType` is the type of this structure.
• `pNext` is `NULL` or a pointer to a structure extending this structure.
• `flags` is reserved for future use.
• `codecQualityPreset` is the Video Decode or Encode quality preset value defined in `VkVideoCodingQualityPresetFlagBitsKHR`.
• `videoSession` is video session object to be bound for the processing of the video commands.
• `videoSessionParameters` is the video session parameters object to be used for the processing of the video commands. If `VK_NULL_HANDLE` is provided here, then no video session parameters apply to this command buffer context.
• `referenceSlotCount` is the number of reference slot entries provided in `pReferenceSlots`.
• `pReferenceSlots` specifies the reference slots used within the video command context between this `vkCmdBeginVideoCodingKHR` command and the `vkCmdEndVideoCodingKHR` command that follows. Each reference slot provides a slot index and the `VkVideoPictureResourceKHR` specifying the reference picture resource bound to this slot index. A slot index `must` not appear more than once in `pReferenceSlots` in a given command.

---

**Valid Usage**

• VUID-VkVideoBeginCodingInfoKHR-referenceSlotCount-04856
  `VkVideoBeginCodingInfoKHR::referenceSlotCount` `must` not exceed the value specified in `VkVideoSessionCreateInfoKHR::maxReferencePicturesSlotsCount` when creating the video session object that is being provided in `videoSession`.

• VUID-VkVideoBeginCodingInfoKHR-videoSessionParameters-04857
  If `videoSessionParameters` is not `VK_NULL_HANDLE`, it `must` have been created using `videoSession` as a parent object.
Valid Usage (Implicit)

- **VUID-VkVideoBeginCodingInfoKHR-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_VIDEO_BEGIN_CODING_INFO_KHR`

- **VUID-VkVideoBeginCodingInfoKHR-pNext-pNext**
  - `pNext` **must** be `NULL`

- **VUID-VkVideoBeginCodingInfoKHR-flags-zerobitmask**
  - `flags` **must** be `0`

- **VUID-VkVideoBeginCodingInfoKHR-codecQualityPreset-parameter**
  - `codecQualityPreset` **must** be a valid combination of `VkVideoCodingQualityPresetFlagBitsKHR` values

- **VUID-VkVideoBeginCodingInfoKHR-codecQualityPreset-requiredbitmask**
  - `codecQualityPreset` **must not** be `0`

- **VUID-VkVideoBeginCodingInfoKHR-videoSession-parameter**
  - `videoSession` **must** be a valid `VkVideoSessionKHR` handle

- **VUID-VkVideoBeginCodingInfoKHR-videoSessionParameters-parameter**
  - If `videoSessionParameters` is not `VK_NULL_HANDLE`, `videoSessionParameters` **must** be a valid `VkVideoSessionParametersKHR` handle

- **VUID-VkVideoBeginCodingInfoKHR-pReferenceSlots-parameter**
  - `pReferenceSlots` **must** be a valid pointer to an array of `referenceSlotCount` valid `VkVideoReferenceSlotKHR` structures

- **VUID-VkVideoBeginCodingInfoKHR-referenceSlotCount-arraylength**
  - `referenceSlotCount` **must** be greater than `0`

- **VUID-VkVideoBeginCodingInfoKHR-videoSessionParameters-parent**
  - If `videoSessionParameters` is a valid handle, it **must** have been created, allocated, or retrieved from `videoSession`

- **VUID-VkVideoBeginCodingInfoKHR-commonparent**
  - Both of `videoSession`, and `videoSessionParameters` that are valid handles of non-ignored parameters **must** have been created, allocated, or retrieved from the same `VkDevice`

The decode preset types are defined with the following:

```c
// Provided by VK_KHR_video_queue
typedef enum VkVideoCodingQualityPresetFlagBitsKHR {    VK_VIDEO_CODING_QUALITY_PRESET_DEFAULT_BIT_KHR = 0,
    VK_VIDEO_CODING_QUALITY_PRESET_NORMAL_BIT_KHR = 0x00000001,
    VK_VIDEO_CODING_QUALITY_PRESET_POWER_BIT_KHR = 0x00000002,
    VK_VIDEO_CODING_QUALITY_PRESET_QUALITY_BIT_KHR = 0x00000004,
} VkVideoCodingQualityPresetFlagBitsKHR;
```

- `VK_VIDEO_CODING_QUALITY_PRESET_NORMAL_BIT_KHR` defines normal decode case.
- `VK_VIDEO_CODING_QUALITY_PRESET_POWER_BIT_KHR` defines power efficient case.
• **VK_VIDEO_CODING_QUALITY_PRESET_QUALITY_BIT_KHR** defines quality focus case.

```c
// Provided by VK_KHR_video_queue
typedef VkFlags VkVideoCodingQualityPresetFlagsKHR;
```

**VkVideoCodingQualityPresetFlagsKHR** is a bitmask type for setting a mask of zero or more **VkVideoCodingQualityPresetFlagBitsKHR**.

The **VkVideoReferenceSlotKHR** structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoReferenceSlotKHR {
    VkStructureType sType;
    const void* pNext;
    int8_t slotIndex;
    const VkVideoPictureResourceKHR* pPictureResource;
} VkVideoReferenceSlotKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **slotIndex** is the unique reference slot index used for the encode or decode operation.
- **pPictureResource** is a pointer to an instance of the **VkVideoPictureResourceKHR** structure describing the picture resource bound to this slot index.

### Valid Usage (Implicit)

- **VUID-VkVideoReferenceSlotKHR-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_VIDEO_REFERENCE_SLOT_KHR**

- **VUID-VkVideoReferenceSlotKHR-pNext-pNext**
  - Each **pNext** member of any structure (including this one) in the **pNext** chain must be either **NULL** or a pointer to a valid instance of **VkVideoDecodeH264DpbSlotInfoEXT** or **VkVideoDecodeH265DpbSlotInfoEXT**

- **VUID-VkVideoReferenceSlotKHR-sType-unique**
  - The **sType** value of each struct in the **pNext** chain must be unique

- **VUID-VkVideoReferenceSlotKHR-pPictureResource-parameter**
  - **pPictureResource** must be a valid pointer to a valid **VkVideoPictureResourceKHR** structure

The **VkVideoPictureResourceKHR** structure is defined as:
typedef struct VkVideoPictureResourceKHR {
    VkStructureType    sType;
    const void*        pNext;
    VkOffset2D         codedOffset;
    VkExtent2D         codedExtent;
    uint32_t           baseArrayLayer;
    VkImageView        imageViewBinding;
} VkVideoPictureResourceKHR;

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **codedOffset** is the offset to be used for the picture resource.
• **codedExtent** is the extent to be used for the picture resource.
• **baseArrayLayer** is the first array layer to be accessed for the Decode or Encode Operations.
• **imageViewBinding** is the image view representing this picture resource.

### Valid Usage (Implicit)

- **VUID-VkVideoPictureResourceKHR-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_VIDEO_PICTURE RESOURCE_KHR**
- **VUID-VkVideoPictureResourceKHR-pNext-pNext**
  - **pNext** must be **NULL**
- **VUID-VkVideoPictureResourceKHR-imageViewBinding-parameter**
  - **imageViewBinding** must be a valid **VkImageView** handle

### 39.4.9. End of the Video Session

To end video decode or encode operations, call:

```c
// Provided by VK_KHR_video_queue
void vkCmdEndVideoCodingKHR(
    VkCommandBuffer                             commandBuffer,
    const VkVideoEndCodingInfoKHR*              pEndCodingInfo);
```

• **commandBuffer** is the command buffer to be filled by this function.
• **pEndCodingInfo** is a pointer to an instance of the **VkVideoEndCodingInfoKHR** structure.
Valid Usage (Implicit)

- VUID-vkCmdEndVideoCodingKHR-commandBuffer-parameter
  
  **commandBuffer** must be a valid **VkCommandBuffer** handle

- VUID-vkCmdEndVideoCodingKHR-pEndCodingInfo-parameter
  
  **pEndCodingInfo** must be a valid pointer to a valid **VkVideoEndCodingInfoKHR** structure

- VUID-vkCmdEndVideoCodingKHR-commandBuffer-recording
  
  **commandBuffer** must be in the recording state

- VUID-vkCmdEndVideoCodingKHR-commandBuffer-cmdpool
  
  The **VkCommandPool** that **commandBuffer** was allocated from must support decode, or encode operations

- VUID-vkCmdEndVideoCodingKHR-renderpass
  
  This command must only be called outside of a render pass instance

- VUID-vkCmdEndVideoCodingKHR-bufferlevel
  
  **commandBuffer** must be a primary **VkCommandBuffer**

Host Synchronization

- Host access to the **VkCommandPool** that **commandBuffer** was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Decode</td>
<td>Encode</td>
</tr>
</tbody>
</table>

The **VkVideoEndCodingInfoKHR** structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoEndCodingInfoKHR {
    VkStructureType             sType;
    const void*                 pNext;
    VkVideoEndCodingFlagsKHR    flags;
} VkVideoEndCodingInfoKHR;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
### 39.4.10. Video Session Control Command

To apply dynamic controls to video decode or video operations, call:

```c
void vkCmdControlVideoCodingKHR(
    VkCommandBuffer commandBuffer,
    const VkVideoCodingControlInfoKHR* pCodingControlInfo);
```

- `commandBuffer` is the command buffer to be filled by this function for setting encode rate control parameters.
- `pCodingControlInfo` is a pointer to `VkVideoCodingControlInfoKHR`.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_VIDEO_END_CODING_INFO_KHR`
- `pNext` must be `NULL`
- `flags` must be `0`
Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Decode</td>
<td>Encode</td>
</tr>
</tbody>
</table>

The `VkVideoCodingControlInfoKHR` structure is defined as:

```c
// Provided by VK_KHR_video_queue
typedef struct VkVideoCodingControlInfoKHR {
    VkStructureType                  sType;
    const void*                      pNext;
    VkVideoCodingControlFlagsKHR     flags;
} VkVideoCodingControlInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.

Valid Usage (Implicit)

- `VUID-VkVideoCodingControlInfoKHR-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_VIDEO_CODING_CONTROL_INFO_KHR`

- `VUID-VkVideoCodingControlInfoKHR-pNext-pNext`  
  `pNext` must be `NULL` or a pointer to a valid instance of `VkVideoEncodeRateControlInfoKHR`

- `VUID-VkVideoCodingControlInfoKHR-sType-unique`  
  The `sType` value of each struct in the `pNext` chain must be unique

- `VUID-VkVideoCodingControlInfoKHR-flags-parameter`  
  `flags` must be a valid combination of `VkVideoCodingControlFlagBitsKHR` values

The `vkCmdControlVideoCodingKHR` flags are defined with the following enumeration:
1. Ensure that the implementation can decode the Video Content by querying the supported codec operations and profiles using `vkGetPhysicalDeviceQueueFamilyProperties2`.

2. By using `vkGetPhysicalDeviceVideoFormatPropertiesKHR` and providing one or more video profiles, choose the Vulkan formats supported by the implementation. The formats for output and reference pictures must be queried and chosen separately. Refer to the section on enumeration of supported video formats.

3. Before creating an image to be used as a video picture resource, obtain the supported image creation parameters by querying with `vkGetPhysicalDeviceFormatProperties2` and `vkGetPhysicalDeviceImageFormatProperties2` using one of the reported formats and adding `VkVideoProfilesKHR` to the pNext chain of `VkFormatProperties2`. When querying the parameters with `vkGetPhysicalDeviceImageFormatProperties2` for images targeting decoded output and reference (Dpb) pictures, the `VkPhysicalDeviceImageFormatInfo2::usage` field should contain `VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR` and `VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR`, respectively.

4. Create none, some, or all of the required images for the decoded output and reference pictures. More Video Picture Resources can be created at some later point if needed while processing the decoded content. Also, if the decoded picture size is expected to change, the images can be created based on the maximum decoded content size required.

5. Create the video session to be used for video decode operations. Before creating the Decode Video Session, the decode capabilities should be queried with `vkGetPhysicalDeviceVideoCapabilitiesKHR` to obtain the limits of the parameters allowed by the implementation for a particular codec profile.

6. Bind memory resources with the decode video session by calling `vkBindVideoSessionMemoryKHR`. The video session cannot be used until memory resources
are allocated and bound to it. In order to determine the required memory sizes and heap types of the device memory allocations, `vkGetVideoSessionMemoryRequirementsKHR` should be called.

7. Create one or more `Session Parameter objects` for use across command buffer recording operations, if required by the codec extension in use. These objects must be created against a `video session` with the parameters required by the codec. Each `Session Parameter object` created is a child object of the associated `Session object` and cannot be bound in the command buffer with any other `Session Object`.

The recording of Video Decode Commands against a Vulkan command buffer consists of the following sequence:

1. `vkCmdBeginVideoCodingKHR` starts the recording of one or more Video Decode operations in the command buffer. For each Video Decode Command operation, a Video Session must be bound to the command buffer within this command. This command establishes a Vulkan Video Decode Context that consists of the bound Video Session Object, Session Parameters Object, and the required Video Picture Resources. The established Video Decode Context is in effect until the `vkCmdEndVideoCodingKHR` command is recorded. If more Video Decode operations are to be required after the `vkCmdEndVideoCodingKHR` command, another Video Decode Context can be started with the `vkCmdBeginVideoCodingKHR` command.

2. `vkCmdDecodeVideoKHR` specifies one or more compressed data buffers to be decoded. The `VkVideoDecodeInfoKHR` parameters, and the codec extension structures chained to this, specify the details of the decode operation.

3. `vkCmdControlVideoCodingKHR` records operations against the decoded data, decoding device, or the Video Session state.

4. `vkCmdEndVideoCodingKHR` signals the end of the recording of the Vulkan Video Decode Context, as established by `vkCmdBeginVideoCodingKHR`.

In addition to the above, the following commands can be recorded between `vkCmdBeginVideoCodingKHR` and `vkCmdEndVideoCodingKHR`:

- Query operations
- Global Memory Barriers
- Buffer Memory Barriers
- Image Memory Barriers (these must be used to transition the Video Picture Resources to the proper `VK_IMAGE_LAYOUT_VIDEO_DECODE_DPB_KHR` and `VK_IMAGE_LAYOUT_VIDEO_DECODE_DST_KHR` layouts).
- Pipeline Barriers
- Events
- Timestamps
- Device Groups (device mask)

The following Video Decode related commands must be recorded outside the Vulkan Video Decode Context established with the `vkCmdBeginVideoCodingKHR` and `vkCmdEndVideoCodingKHR` commands:
39.5.1. Video Picture Decode Modes

There are a few ways that the `vkCmdDecodeVideoKHR` can be configured for the Video Picture Decode Operations, based on:

- if the output resource would need to be used as Reference Picture for subsequent decode operations and;
- if Dpb slots are required for use as Reference Pictures indexes.

The most basic Video Picture Decode operation with the `vkCmdDecodeVideoKHR` command is to output the decoded pixel data without using any Dpb Reference Pictures and without updating any Dpb slot's indexes.

In this case, the following `VkVideoDecodeInfoKHR` parameters must be set:

- `VkVideoDecodeInfoKHR::pSetupReferenceSlot->pPictureResource->imageViewBinding` must be a valid `VkImageView`. This `VkImageView` represents the output resource where the decoded pixels will be populated after a successful decode operation.
- `VkVideoDecodeInfoKHR::pSetupReferenceSlot->slotIndex` must be an invalid Dpb slot index (-1) since the decoded picture is not intended to be used as a reference picture with subsequent video decode operations.
- The value of the `VkVideoDecodeInfoKHR::referenceSlotCount` can be zero and `VkVideoDecodeInfoKHR::pReferenceSlots` can be NULL.
- If `VkVideoDecodeInfoKHR::pReferenceSlots` is not NULL, it can still have entries representing Dpb slot indexes with a Valid Picture Reference. The codec extension selects the actual use of the Reference Pictures by referring to a Dpb slot index with a Valid Picture Reference.

Video Picture Decode operations with the `vkCmdDecodeVideoKHR` command, requiring one or more Reference Pictures for the predictions of the values of samples for the decoded output picture would require Dpb slots with Valid Picture Reference.

In this case, the following `VkVideoDecodeInfoKHR` parameters must be set:

- `VkVideoDecodeInfoKHR::pSetupReferenceSlot->pPictureResource->imageViewBinding` must be a valid `VkImageView`. This `VkImageView` represents the output resource where the decoded pixels will be populated after a successful decode operation.
- `VkVideoDecodeInfoKHR::pSetupReferenceSlot->slotIndex` must be an invalid Dpb slot index (-1) since the decoded picture is not intended to be used as a reference picture with subsequent video decode operations.
- The value of the `VkVideoDecodeInfoKHR::referenceSlotCount` must not be zero and `VkVideoDecodeInfoKHR::pReferenceSlots` should represent at least the number of the reference slots required for the decode operation. The codec extension selects the actual use of the
**Reference Pictures** by referring to a Dpb slot index with a **Valid Picture Reference**. If the implementation does not use an opaque Dpb, each Dpb slot representing a reference picture must refer to a valid image view. The image views must represent the same image resources that were used to create the reference picture for the corresponding Dpb slot index.

- **VkVideoDecodeInfoKHR::pReferenceSlots** can still have entries representing Dpb slot indexes with a **Valid Picture Reference**.

After the **vkCmdDecodeVideoKHR** operation is completed successfully, the **VkVideoDecodeInfoKHR::pSetupReferenceSlot->pPictureResource->imageViewBinding** pixel data will be updated with the decoded content. The operation will not update any Dpb slot with **Reference Pictures** data. However, any Dpb slot activation, invalidation, or deactivation operations requested via **VkVideoDecodeInfoKHR::pReferenceSlots** are still going to be performed.

![Diagram](image.png)

*Figure 28. Decoding a Frame to VkImageView without a slot update for a Reference Picture*

**Video Picture Decode with a Reference Picture slot update and using optional Reference Pictures**

When it is known that the picture to be decoded will be used as a reference picture for subsequent decode operations, one of the available Dpb slots needs to be selected for activation and update operations as part of the **vkCmdDecodeVideoKHR** command.

Based on whether a decode operation with reference pictures or without reference pictures is required, the **vkCmdDecodeVideoKHR** should be configured with parameters as described in the previous sections. In addition, one of the available Dpb slots must be selected by the application, activated with resources and then set-up for an update with the decode operation.

In this case, the following **VkVideoDecodeInfoKHR** parameters must be set:

- **VkVideoDecodeInfoKHR::pSetupReferenceSlot->pPictureResource->imageViewBinding** must be a valid VkImageView. This VkImageView represents the output resource where the decoded pixels will be populated after a successful decode operation. If the implementation does not use an opaque Dpb, both the output and reference picture resource coincide.
• **VkVideoDecodeInfoKHR::pSetupReferenceSlot->slotIndex** must be a valid Dpb slot index selected by the application, based on the currently available slots.

• **VkVideoDecodeInfoKHR::pReferenceSlots** can still have entries representing Dpb slot indexes with a Valid Picture Reference.

After the **vkCmdDecodeVideoKHR** operation has completed successfully, the decoded content will be available in the resource provided for **VkVideoDecodeInfoKHR::pSetupReferenceSlot->pNext->imageViewBinding**. In addition, this operation will update the selected Dpb slot with Reference Pictures data. Any other Dpb slot activation, invalidation, or deactivation operation requested via the **VkVideoDecodeInfoKHR::pReferenceSlots** are going to be performed as well.

![Diagram](image-url)

**Figure 29. Decoding a Frame to VkImageView with an update to a Reference Pictures Dpb slot**

### 39.5.2. Video Decode Command Buffer Commands

To decode a frame, call:

```c
// Provided by VK_KHR_video_decode_queue
void vkCmdDecodeVideoKHR(
    VkCommandBuffer                             commandBuffer,
    const VkVideoDecodeInfoKHR*                 pFrameInfo);
```

- **commandBuffer** is the command buffer to be filled with this function for decode frame command.
- **pFrameInfo** is the pointer of **VkVideoDecodeInfoKHR**.
Valid Usage (Implicit)

- VUID-vkCmdDecodeVideoKHR-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdDecodeVideoKHR-pFrameInfo-parameter
  `pFrameInfo` must be a valid pointer to a valid `VkVideoDecodeInfoKHR` structure

- VUID-vkCmdDecodeVideoKHR-commandBuffer-recording
  `commandBuffer` must be in the recording state

- VUID-vkCmdDecodeVideoKHR-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support decode operations

- VUID-vkCmdDecodeVideoKHR-renderpass
  This command must only be called outside of a render pass instance

- VUID-vkCmdDecodeVideoKHR-bufferlevel
  `commandBuffer` must be a primary `VkCommandBuffer`

Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Decode</td>
<td></td>
</tr>
</tbody>
</table>

The `VkVideoDecodeInfoKHR` structure is defined as:
// Provided by VK_KHR_video_decode_queue

typedef struct VkVideoDecodeInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkVideoDecodeFlagsKHR flags;
    VkOffset2D codedOffset;
    VkExtent2D codedExtent;
    VkBuffer srcBuffer;
    VkDeviceSize srcBufferOffset;
    VkDeviceSize srcBufferRange;
    VkVideoPictureResourceKHR dstPictureResource;
    const VkVideoReferenceSlotKHR* pSetupReferenceSlot;
    uint32_t referenceSlotCount;
    const VkVideoReferenceSlotKHR* pReferenceSlots;
} VkVideoDecodeInfoKHR;

- **sType** is the type of this structure.

- **pNext** is NULL or a pointer to a structure extending this structure. All the codec specific structures related to each frame (picture parameters, quantization matrix, etc.) must be chained here and pass to decode session with the function call `vkCmdDecodeVideoKHR`.

- **flags** are decode flags of type `VkVideoDecodeFlagBitsKHR`, reserved for future versions of this specification.

- **codedOffset** is the coded offset of the decode operations. The purpose of this field is interpreted based on the codec extension. When decoding content in H.264 field mode, the `codedOffset` specifies the line or picture field's offset within the image.

- **codedExtent** is the coded size of the decode operations.

- **srcBuffer** is the source buffer that holds the encoded bitstream.

- **srcBufferOffset** is the buffer offset where the valid encoded bitstream starts in `srcBuffer`. It must meet the alignment requirement `minBitstreamBufferOffsetAlignment` within `VkVideoCapabilitiesKHR` queried with the `vkGetPhysicalDeviceVideoCapabilitiesKHR` function.

- **srcBufferRange** is the size of the `srcBuffer` with valid encoded bitstream, starting from `srcBufferOffset`. It must meet the alignment requirement `minBitstreamBufferSizeAlignment` within `VkVideoCapabilitiesKHR` queried with the `vkGetPhysicalDeviceVideoCapabilitiesKHR` function.

- **dstPictureResource** is the destination Decoded Output Picture Resource.

- **pSetupReferenceSlot** is an optional parameter used for generating a Dpb reference slot and Picture Resource. `VkVideoReferenceSlotKHR::slotIndex` specifies the slot index number to use as a target for producing the Dpb data. `slotIndex` must reference a valid entry as specified in `VkVideoBeginCodingInfoKHR` via the `pReferenceEntries` within the `vkCmdBeginVideoCodingKHR` command that established the Vulkan Video Decode Context for this command.

- **referenceSlotCount** is the number of the Dpb Reference Pictures that will be used when this decoding operation is executing.
• **pReferenceSlots** is a list of the Dpb Reference pictures that will be used when this decoding operation is executing.

### Valid Usage (Implicit)

- **VUID-VkVideoDecodeInfoKHR-sType-sType**  
  *sType* must be `VK_STRUCTURE_TYPE_VIDEO_DECODE_INFO_KHR`

- **VUID-VkVideoDecodeInfoKHR-pNext-pNext**  
  Each *pNext* member of any structure (including this one) in the *pNext* chain must be either **NULL** or a pointer to a valid instance of `VkVideoDecodeH264PictureInfoEXT` or `VkVideoDecodeH265PictureInfoEXT`

- **VUID-VkVideoDecodeInfoKHR-sType-unique**  
  The *sType* value of each struct in the *pNext* chain must be unique

- **VUID-VkVideoDecodeInfoKHR-flags-parameter**  
  *flags* must be a valid combination of `VkVideoDecodeFlagBitsKHR` values

- **VUID-VkVideoDecodeInfoKHR-srcBuffer-parameter**  
  *srcBuffer* must be a valid `VkBuffer` handle

- **VUID-VkVideoDecodeInfoKHR-dstPictureResource-parameter**  
  *dstPictureResource* must be a valid `VkVideoPictureResourceKHR` structure

- **VUID-VkVideoDecodeInfoKHR-pSetupReferenceSlot-parameter**  
  *pSetupReferenceSlot* must be a valid pointer to a valid `VkVideoReferenceSlotKHR` structure

- **VUID-VkVideoDecodeInfoKHR-pReferenceSlots-parameter**  
  *pReferenceSlots* must be a valid pointer to an array of `referenceSlotCount` valid `VkVideoReferenceSlotKHR` structures

- **VUID-VkVideoDecodeInfoKHR-referenceSlotCount-arraylength**  
  *referenceSlotCount* must be greater than 0

The **vkCmdDecodeVideoKHR** flags are defined with the following enumeration:

```c
// Provided by VK_KHR_video_decode_queue  
typedef enum VkVideoDecodeFlagBitsKHR {  
    VK_VIDEO_DECODE_DEFAULT_KHR = 0,  
    VK_VIDEO_DECODE_RESERVED_0_BIT_KHR = 0x00000001,  
} VkVideoDecodeFlagBitsKHR;
```

- **VK_VIDEO_DECODE_RESERVED_0_BIT_KHR** The current version of the specification has reserved this value for future use.

```c
// Provided by VK_KHR_video_decode_queue  
typedef VkFlags VkVideoDecodeFlagsKHR;
```
VkVideoDecodeFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoDecodeFlagBitsKHR.

39.6. Video Decode of AVC (ITU-T H.264)

This extension adds H.264 codec specific structures needed for decode session to execute decode jobs which include H.264 sequence header, picture parameter header and quantization matrix etc. Unless otherwise noted, all references to the H.264 specification are to the 2010 edition published by the ITU-T, dated March 2010. This specification is available at http://www.itu.int/rec/T-REC-H.264.

39.6.1. H.264 decode profile

A H.264 decode profile is specified using VkVideoDecodeH264ProfileEXT chained to VkVideoProfileKHR when the codec-operation in VkVideoProfileKHR is VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_EXT.

The VkVideoDecodeH264ProfileEXT structure is defined as:

```c
// Provided by VK_EXT_video_decode_h264
typedef struct VkVideoDecodeH264ProfileEXT {
    VkStructureType sType;
    const void* pNext;
    StdVideoH264ProfileIdc stdProfileIdc;
    VkVideoDecodeH264FieldLayoutFlagsEXT fieldLayout;
} VkVideoDecodeH264ProfileEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `stdProfileIdc` is the H.264 codec profile IDC defined in StdVideoH264ProfileIdc for enumeration.
- `fieldLayout` is the field layout for interlaced content as defined in VkVideoDecodeH264FieldLayoutFlagBitsEXT for enumeration.

**Valid Usage (Implicit)**

- VUID-VkVideoDecodeH264ProfileEXT-sType-sType
  `sType must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_PROFILE_EXT`

- VUID-VkVideoDecodeH264ProfileEXT-fieldLayout-parameter
  `fieldLayout must be a valid combination of VkVideoDecodeH264FieldLayoutFlagBitsEXT values`

- VUID-VkVideoDecodeH264ProfileEXT-fieldLayout-required bitmask
  `fieldLayout must not be 0`

The H.264 decode interlaced field layout flags are defined with the following enum:
typedef enum VkVideoDecodeH264FieldLayoutFlagBitsEXT {
    VK_VIDEO_DECODE_H264_PROGRESSIVE_PICTURES_ONLY_EXT = 0,
    VK_VIDEO_DECODE_H264_FIELD_LAYOUT_LINE_INTERLACED_PLANE_BIT_EXT = 0x00000001,
    VK_VIDEO_DECODE_H264_FIELD_LAYOUT_SEPARATE_INTERLACED_PLANE_BIT_EXT = 0x00000002,
} VkVideoDecodeH264FieldLayoutFlagBitsEXT;

- VK_VIDEO_DECODE_H264_FIELD_LAYOUT_LINE_INTERLACED_PLANE_BIT_EXT - Interlaced content is supported where each field is decoded within the even/odd line of a picture resource.
- VK_VIDEO_DECODE_H264_FIELD_LAYOUT_SEPARATE_INTERLACED_PLANE_BIT_EXT
  - Interlaced content is supported where each field is decoded within a separate picture plane or resource.

### 39.6.2. Selecting a H.264 decode profile

When using `vkGetPhysicalDeviceVideoCapabilitiesKHR` to query the capabilities for the input `pVideoProfile` with `videoCodecOperation` specified as `VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_EXT`, the instance of `VkVideoDecodeH264ProfileEXT` structure must be chained to `VkVideoProfileKHR` to select a H.264 decode profile. If supported, the implementation returns the capabilities associated with the specified H.264 decode profile. The requirement is similar when querying supported image formats using `vkGetPhysicalDeviceVideoFormatPropertiesKHR`.

A supported H.264 decode profile must be selected when creating a video session by chaining `VkVideoDecodeH264ProfileEXT` to the `VkVideoProfileKHR` field of `VkVideoSessionCreateInfoKHR`.

### 39.6.3. Capabilities

When using `vkGetPhysicalDeviceVideoCapabilitiesKHR` to query the capabilities for the input `pVideoProfile` with `videoCodecOperation` specified as `VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_EXT`, the instance of `VkVideoDecodeH264CapabilitiesEXT` structure must be chained to `VkVideoCapabilitiesKHR` to get this H.264 decode profile specific capabilities.

The `VkVideoDecodeH264CapabilitiesEXT` structure is defined as:

```c
typedef struct VkVideoDecodeH264CapabilitiesEXT {
    VkStructureType          sType;
    void*                    pNext;
    uint32_t                 maxLevel;
    VkOffset2D               fieldOffsetGranularity;
    VkExtensionProperties    stdExtensionVersion;
} VkVideoDecodeH264CapabilitiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxLevel` is the maximum H.264 level supported by the device.
• **fieldOffsetGranularity** - if Interlaced Video Content is supported, the maximum field offset granularity supported for the picture resource.

• **stdExtensionVersion** is the specific H.264 extension name and version supported by this implementation.

### Valid Usage (Implicit)

```
• VUID-VkVideoDecodeH264CapabilitiesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODER_H264_CAPABILITIES_EXT
```

### 39.6.4. Create Information

The instance of **VkVideoDecodeH264SessionCreateInfoEXT** structure can be chained to **VkVideoSessionCreateInfoKHR** when the function **vkCreateVideoSessionKHR** is called to create a video session for H.264 decode.

The **VkVideoDecodeH264SessionCreateInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_video_decode_h264
typedef struct VkVideoDecodeH264SessionCreateInfoEXT {
    VkStructureType                    sType;  
    const void*                        pNext;  
    VkVideoDecodeH264CreateFlagsEXT    flags;  
    const VkExtensionProperties*       pStdExtensionVersion;
} VkVideoDecodeH264SessionCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is reserved for future use.
- **pStdExtensionVersion** is the specific H.264 codec extensions defined in StdVideoH264Extensions.

### Valid Usage (Implicit)

```
• VUID-VkVideoDecodeH264SessionCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODER_H264_SESSION_CREATE_INFO_EXT
• VUID-VkVideoDecodeH264SessionCreateInfoEXT-flags-zero bitmask
  flags must be 0
• VUID-VkVideoDecodeH264SessionCreateInfoEXT-pStdExtensionVersion-parameter
  pStdExtensionVersion must be a valid pointer to a valid VkExtensionProperties structure
```

### 39.6.5. Decoder Parameter Sets

To reduce parameter traffic during decoding, the decoder parameter set object supports storing
H.264 SPS/PPS parameter sets that **may** be later referenced during decoding.

An instance of `VkVideoDecodeH264SessionParametersCreateInfoEXT` holding one H.264 SPS and at least one H.264 PPS parameter set **must** be chained to `VkVideoSessionParametersCreateInfoKHR` when calling `vkCreateVideoSessionParametersKHR` to store these parameter set(s) with the decoder parameter set object for later reference. The provided H.264 SPS/PPS parameters **must** be within the limits specified during decoder creation for the decoder specified in `VkVideoSessionParametersCreateInfoKHR`.

The `VkVideoDecodeH264SessionParametersCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_video_decode_h264
typedef struct VkVideoDecodeH264SessionParametersCreateInfoEXT {
    VkStructureType                                        sType;
    const void*                                            pNext;
    uint32_t                                               maxSpsStdCount;
    uint32_t                                               maxPpsStdCount;
    const VkVideoDecodeH264SessionParametersAddInfoEXT*    pParametersAddInfo;
} VkVideoDecodeH264SessionParametersCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **maxSpsStdCount** is the maximum number of SPS parameters that the `VkVideoSessionParametersKHR` can contain.
- **maxPpsStdCount** is the maximum number of PPS parameters that the `VkVideoSessionParameters_KHR` can contain.
- **pParametersAddInfo** is a pointer to an optional set of H.264 parameters to add upon object creation.

### Valid Usage ( Implicit )

- **VUID-VkVideoDecodeH264SessionParametersCreateInfoEXT-sType-sType**
  
  **sType** **must** be `VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_PARAMETERS_CREATE_INFO_EXT`

- **VUID-VkVideoDecodeH264SessionParametersCreateInfoEXT-pParametersAddInfo-parameter**
  
  If `pParametersAddInfo` is not `NULL`, `pParametersAddInfo` **must** be a valid pointer to a valid `VkVideoDecodeH264SessionParametersAddInfoEXT` structure
// Provided by VK_EXT_video_decode_h264

typedef struct VkVideoDecodeH264SessionParametersAddInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t spsStdCount;
    const StdVideoH264SequenceParameterSet* pSpsStd;
    uint32_t ppsStdCount;
    const StdVideoH264PictureParameterSet* pPpsStd;
} VkVideoDecodeH264SessionParametersAddInfoEXT;

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **spsStdCount** is the number of SPS elements in the **pSpsStd**. Its value **must** be less than or equal to the value of **maxSpsStdCount**.
• **pSpsStd** is an array of structures representing H.264 sequence parameter sets. Each StdVideoH264SequenceParameterSet entry **must** have a unique H.264 SPS ID.
• **ppsStdCount** is the number of PPS provided in **pPpsStd**. Its value **must** be less than or equal to the value of **maxPpsStdCount**.
• **pPpsStd** is an array of structures representing H.264 picture parameter sets. Each StdVideoH264PictureParameterSet entry **must** have a unique H.264 SPS-PPS ID pair.
Valid Usage

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-spsStdCount-04822**
  The values of `spsStdCount` and `ppsStdCount` must be less than or equal to the values of `maxSpsStdCount` and `maxPpsStdCount`, respectively.

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-maxSpsStdCount-04823**
  When the `maxSpsStdCount` number of parameters of type `StdVideoH264SequenceParameterSet` in the Video Session Parameters object is reached, no additional parameters of that type can be added to this object. `VK_ERROR_TOO_MANY_OBJECTS` will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-maxPpsStdCount-04824**
  When the `maxPpsStdCount` number of parameters of type `StdVideoH264PictureParameterSet` in the Video Session Parameters object is reached, no additional parameters of that type can be added to this object. `VK_ERROR_TOO_MANY_OBJECTS` will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-None-04825**
  Each entry to be added must have a unique, to the rest of the parameter array entries and the existing parameters in the Video Session Parameters Object that is being updated, SPS-PPS IDs.

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-None-04826**
  Parameter entries that already exist in Video Session Parameters object with a particular SPS-PPS IDs cannot be replaced nor updated.

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-None-04827**
  When creating a new object using a Video Session Parameters as a template, the array's parameters with the same SPS-PPS IDs as the ones from the template take precedence.

- **VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-None-04828**
  SPS/PPS parameters must comply with the limits specified in `VkVideoSessionCreateInfoKHR` during Video Session creation.
Valid Usage (Implicit)

- VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-sType-sType
  
  must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_PARAMETERS_ADD_INFO_EXT

- VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-pSpsStd-parameter
  
  If pSpsStd is not NULL, pSpsStd must be a valid pointer to an array of spsStdCount StdVideoH264SequenceParameterSet values

- VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-pPpsStd-parameter
  
  If pPpsStd is not NULL, pPpsStd must be a valid pointer to an array of ppsStdCount StdVideoH264PictureParameterSet values

- VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-spsStdCount-arraylength
  
  spsStdCount must be greater than 0

- VUID-VkVideoDecodeH264SessionParametersAddInfoEXT-ppsStdCount-arraylength
  
  ppsStdCount must be greater than 0

39.6.6. Picture Decoding

To decode a picture, the structure VkVideoDecodeH264PictureInfoEXT may be chained to VkVideoDecodeInfoKHR when calling vkCmdDecodeVideoKHR.

The structure VkVideoDecodeH264PictureInfoEXT representing a picture decode operation is defined as:

```c
// Provided by VK_EXT_video_decode_h264
typedef struct VkVideoDecodeH264PictureInfoEXT {
    VkStructureType sType;
    const void* pNext;
    const StdVideoDecodeH264PictureInfo* pStdPictureInfo;
    uint32_t slicesCount;
    const uint32_t* pSlicesDataOffsets;
} VkVideoDecodeH264PictureInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- pStdPictureInfo is a pointer to the codec standard specific picture information from H.264 spec.
- slicesCount is the number of slices in this picture.
- pSlicesDataOffsets is an array of slicesCount offsets indicating the start offset of each slice within the bitstream buffer.
Valid Usage (Implicit)

- VUID-VkVideoDecodeH264PictureInfoEXT-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_PICTURE_INFO_EXT

- VUID-VkVideoDecodeH264PictureInfoEXT-pStdPictureInfo-parameter
  
  pStdPictureInfo must be a valid pointer to a valid StdVideoDecodeH264PictureInfo value

- VUID-VkVideoDecodeH264PictureInfoEXT-pSlicesDataOffsets-parameter
  
  pSlicesDataOffsets must be a valid pointer to an array of slicesCount uint32_t values

- VUID-VkVideoDecodeH264PictureInfoEXT-slicesCount-arraylength
  
  slicesCount must be greater than 0

The VkVideoDecodeH264DpbSlotInfoEXT structure correlates a DPB slot index with codec-specific information and is defined as:

```c
// Provided by VK_EXT_video_decode_h264
typedef struct VkVideoDecodeH264DpbSlotInfoEXT {
    VkStructureType                          sType;
    const void*                              pNext;
    const StdVideoDecodeH264ReferenceInfo*    pStdReferenceInfo;
} VkVideoDecodeH264DpbSlotInfoEXT;
```

- sType is the type of this structure.
- pStdReferenceInfo is a pointer to the codec standard specific picture reference information from H.264 spec.

Valid Usage (Implicit)

- VUID-VkVideoDecodeH264DpbSlotInfoEXT-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_DPB_SLOT_INFO_EXT

- VUID-VkVideoDecodeH264DpbSlotInfoEXT-pStdReferenceInfo-parameter
  
  pStdReferenceInfo must be a valid pointer to a valid StdVideoDecodeH264ReferenceInfo value

The VkVideoDecodeH264MvcEXT structure is defined as:

```c
// Provided by VK_EXT_video_decode_h264
typedef struct VkVideoDecodeH264MvcEXT {
    VkStructureType                          sType;
    const void*                              pNext;
    const StdVideoDecodeH264Mvc*             pStdMvc;
} VkVideoDecodeH264MvcEXT;
```

- sType is the type of this structure.
• pNext is NULL or a pointer to an extension specific structure.
• pStdMvc is a pointer to H.264 codec spec information for MVC.

When the content type is H.264 MVC, an instance of VkVideoDecodeH264MvcEXT must be chained to VkVideoDecodeH264PictureInfoEXT.

Valid Usage (Implicit)

• VUID-VkVideoDecodeH264MvcEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_MVC_EXT
• VUID-VkVideoDecodeH264MvcEXT-pStdMvc-parameter
  pStdMvc must be a valid pointer to a valid StdVideoDecodeH264Mvc value

39.7. Video Decode of HEVC (ITU-T H.265)

This extension adds H.265 codec specific structures needed for decode session to execute decode jobs which include H.265 sequence header, picture parameter header and quantization matrix etc. Unless otherwise noted, all references to the H.265 specification are to the 2013 edition published by the ITU-T, dated April 2013. This specification is available at http://www.itu.int/rec/T-REC-H.265.

39.7.1. H.265 decode profile

A H.265 decode profile is specified using VkVideoDecodeH265ProfileEXT chained to VkVideoProfileKHR when the codec-operation in VkVideoProfileKHR is VK_VIDEO_CODEC_OPERATION_DECODE_H265_BIT_EXT.

The VkVideoDecodeH265ProfileEXT structure is defined as:

```c
// Provided by VK_EXT_video_decode_h265
typedef struct VkVideoDecodeH265ProfileEXT {
    VkStructureType           sType;
    const void*               pNext;
    StdVideoH265ProfileIdc    stdProfileIdc;
} VkVideoDecodeH265ProfileEXT;
```

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• stdProfileIdc is the H.265 codec profile IDC defined in StdVideoH265ProfileIdc for enumeration.

Valid Usage (Implicit)

• VUID-VkVideoDecodeH265ProfileEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_PROFILE_EXT
39.7.2. Selecting an H.265 Profile

When using `vkGetPhysicalDeviceVideoCapabilitiesKHR` to query the capabilities for the input `pVideoProfile` with `videoCodecOperation` specified as `VK_VIDEO_CODEC_OPERATION_DECODE_H265_BIT_EXT`, the instance of `VkVideoDecodeH265ProfileEXT` structure must be chained to `VkVideoProfileKHR` to select a H.265 decode profile. If supported, the implementation returns the capabilities associated with the specified H.265 decode profile. The requirement is similar when querying supported image formats using `vkGetPhysicalDeviceVideoFormatPropertiesKHR`.

A supported H.265 decode profile must be selected when creating a video session by chaining `VkVideoDecodeH265ProfileEXT` to the `VkVideoProfileKHR` field of `VkVideoSessionCreateInfoKHR`.

39.7.3. Capabilities

When uses `vkGetPhysicalDeviceVideoCapabilitiesKHR` to query the capabilities for the parameter `videoCodecOperation` specified as `VK_VIDEO_CODEC_OPERATION_DECODE_H265_BIT_EXT`, `VkVideoDecodeH265CapabilitiesEXT` structure can be chained to `VkVideoCapabilitiesKHR` to get this H.265 extension specific capabilities.

The `VkVideoDecodeH265CapabilitiesEXT` structure is defined as:

```c
// Provided by VK_EXT_video_decode_h265
typedef struct VkVideoDecodeH265CapabilitiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxLevel;
    VkExtensionProperties stdExtensionVersion;
} VkVideoDecodeH265CapabilitiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `maxLevel` is the maximum H.265 level supported by the device.
- `stdExtensionVersion` is the specific H.265 extension name and version supported by this implementation.

```
Valid Usage (Implicit)
```

- `VUID-VkVideoDecodeH265CapabilitiesEXT-sType-sType` `sType must be VK_STRUCTURE_TYPE_VIDEO_DECODER_H265_CAPABILITIES_EXT`

39.7.4. Create Information

The instance of `VkVideoDecodeH265SessionCreateInfoEXT` structure can be chained to `VkVideoSessionCreateInfoKHR` when the function `vkCreateVideoSessionKHR` is called to create a video session for H.265 decode operations.
The `VkVideoDecodeH265SessionCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_video_decode_h265
typedef struct VkVideoDecodeH265SessionCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkVideoDecodeH265CreateFlagsEXT flags;
    const VkExtensionProperties* pStdExtensionVersion;
} VkVideoDecodeH265SessionCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is reserved for future use.
- `pStdExtensionVersion` is the specific H.265 codec extensions defined in `StdVideoH265Extensions`.

**Valid Usage (Implicit)**

- VUID-VkVideoDecodeH265SessionCreateInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_CREATE_INFO_EXT`
- VUID-VkVideoDecodeH265SessionCreateInfoEXT-flags-zerobitmask
  `flags` must be `0`
- VUID-VkVideoDecodeH265SessionCreateInfoEXT-pStdExtensionVersion-parameter
  `pStdExtensionVersion` must be a valid pointer to a valid `VkExtensionProperties` structure

### 39.7.5. Decoder Parameter Sets

To reduce parameter traffic during decoding, the decoder parameter set object supports storing H.265 SPS/PPS parameter sets that may be later referenced during decoding.

An instance of `VkVideoDecodeH265SessionParametersCreateInfoEXT` holding one H.265 SPS and at least one H.265 PPS parameter set must be chained to `VkVideoSessionParametersCreateInfoKHR` when calling `vkCreateVideoSessionParametersKHR` to store these parameter set(s) with the decoder parameter set object for later reference. The provided H.265 SPS/PPS parameters must be within the limits specified during decoder creation for the decoder specified in `VkVideoSessionParametersCreateInfoKHR`.

The `VkVideoDecodeH265SessionParametersCreateInfoEXT` structure is defined as:
typedef struct VkVideoDecodeH265SessionParametersCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t maxSpsStdCount;
    uint32_t maxPpsStdCount;
    const VkVideoDecodeH265SessionParametersAddInfoEXT* pParametersAddInfo;
} VkVideoDecodeH265SessionParametersCreateInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxSpsStdCount** is the maximum number of SPS parameters that the VkVideoSessionParametersKHR can contain.
- **maxPpsStdCount** is the maximum number of PPS parameters that the VkVideoSessionParametersKHR can contain.
- **pParametersAddInfo** is a pointer to an optional set of H.265 parameters to add upon object creation.

Valid Usage (Implicit)

- **VUID-VkVideoDecodeH265SessionParametersCreateInfoEXT-sType-sType**
  sType must be VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_PARAMETERS_CREATE_INFO_EXT

- **VUID-VkVideoDecodeH265SessionParametersCreateInfoEXT-pParametersAddInfo-parameter**
  If pParametersAddInfo is not NULL, pParametersAddInfo must be a valid pointer to a valid VkVideoDecodeH265SessionParametersAddInfoEXT structure

typedef struct VkVideoDecodeH265SessionParametersAddInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t spsStdCount;
    const StdVideoH265SequenceParameterSet* pSpsStd;
    uint32_t ppsStdCount;
    const StdVideoH265PictureParameterSet* pPpsStd;
} VkVideoDecodeH265SessionParametersAddInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **spsStdCount** is the number of SPS elements in the pSpsStd. Its value must be less than or equal to the value of maxSpsStdCount.
- **pSpsStd** is an array of structures representing H.265 sequence parameter sets. Each StdVideoH265SequenceParameterSet entry must have a unique H.265 VPS-SPS ID pair.
• **ppsStdCount** is the number of PPS provided in **pPpsStd**. Its value **must** be less than or equal to the value of **maxPpsStdCount**.

• **pPpsStd** is an array of structures representing H.265 picture parameter sets. Each StdVideoH265PictureParameterSet entry **must** have a unique H.265 VPS-SPS-PPS ID tuple.

## Valid Usage

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-vpsStdCount-04829**
  The values of **vpsStdCount**, **spsStdCount** and **ppsStdCount** **must** be less than or equal to the values of **maxVpsStdCount**, **maxSpsStdCount** and **maxPpsStdCount**, respectively.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-maxVpsStdCount-04830**
  When the **maxVpsStdCount** number of parameters of type StdVideoH265VideoParameterSet in the Video Session Parameters object is reached, no additional parameters of that type can be added to the object. **VK_ERROR_TOO_MANY_OBJECTS** will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-maxSpsStdCount-04831**
  When the **maxSpsStdCount** number of parameters of type StdVideoH265SequenceParameterSet in the Video Session Parameters object is reached, no additional parameters of that type can be added to the object. **VK_ERROR_TOO_MANY_OBJECTS** will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-maxPpsStdCount-04832**
  When the **maxPpsStdCount** number of parameters of type StdVideoH265PictureParameterSet in the Video Session Parameters object is reached, no additional parameters of that type can be added to the object. **VK_ERROR_TOO_MANY_OBJECTS** will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-None-04833**
  Each entry to be added **must** have a unique, to the rest of the parameter array entries and the existing parameters in the Video Session Parameters Object that is being updated, VPS-SPS-PPS IDs.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-None-04834**
  Parameter entries that already exist in Video Session Parameters object with a particular VPS-SPS-PPS IDs **cannot** be replaced nor updated.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-None-04835**
  When creating a new object using a Video Session Parameters as a template, the array’s parameters with the same VPS-SPS-PPS IDs as the ones from the template take precedence.

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-None-04836**
  VPS/SPS/PPS parameters **must** comply with the limits specified in **VkVideoSessionCreateInfoKHR** during Video Session creation.
Valid Usage (Implicit)

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-sType-sType**
  *sType* must be `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_PARAMETERS_ADD_INFO_EXT`

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-pSpsStd-parameter**
  If `pSpsStd` is not NULL, `pSpsStd` must be a valid pointer to an array of `spsStdCount` `StdVideoH265SequenceParameterSet` values

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-pPpsStd-parameter**
  If `pPpsStd` is not NULL, `pPpsStd` must be a valid pointer to an array of `ppsStdCount` `StdVideoH265PictureParameterSet` values

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-spsStdCount-arraylength**
  `spsStdCount` must be greater than 0

- **VUID-VkVideoDecodeH265SessionParametersAddInfoEXT-ppsStdCount-arraylength**
  `ppsStdCount` must be greater than 0

39.7.6. Picture Parameters

The `VkVideoDecodeH265PictureInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_video_decode_h265
typedef struct VkVideoDecodeH265PictureInfoEXT {
    VkStructureType                   sType;
    const void*                       pNext;
    StdVideoDecodeH265PictureInfo*    pStdPictureInfo;
    uint32_t                          slicesCount;
    const uint32_t*                   pSlicesDataOffsets;
} VkVideoDecodeH265PictureInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **pStdPictureInfo** is the codec standard specific structure from H.265 spec.
- **slicesCount** is the number of slices in this picture.
- **pSlicesDataOffsets** is an array of `slicesCount` offsets indicating the start offset of each slice within the bitstream buffer.
Valid Usage (Implicit)

- VUID-VkVideoDecodeH265PictureInfoEXT-sType-sType
  
  **sType must** be **VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_PICTURE_INFO_EXT**

- VUID-VkVideoDecodeH265PictureInfoEXT-pStdPictureInfo-parameter
  
  **pStdPictureInfo must** be a valid pointer to a **StdVideoDecodeH265PictureInfo** value

- VUID-VkVideoDecodeH265PictureInfoEXT-pSlicesDataOffsets-parameter
  
  **pSlicesDataOffsets must** be a valid pointer to an array of **slicesCount uint32_t** values

- VUID-VkVideoDecodeH265PictureInfoEXT-slicesCount-arraylength
  
  **slicesCount must** be greater than 0

The **VkVideoDecodeH265DpbSlotInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_video_decode_h265
typedef struct VkVideoDecodeH265DpbSlotInfoEXT {
    VkStructureType                           sType;
    const void*                               pNext;
    const StdVideoDecodeH265ReferenceInfo*    pStdReferenceInfo;
} VkVideoDecodeH265DpbSlotInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **pStdReferenceInfo** is a pointer to the codec standard specific picture reference information from H.264 spec.

Valid Usage (Implicit)

- VUID-VkVideoDecodeH265DpbSlotInfoEXT-sType-sType
  
  **sType must** be **VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_DPB_SLOT_INFO_EXT**

- VUID-VkVideoDecodeH265DpbSlotInfoEXT-pStdReferenceInfo-parameter
  
  **pStdReferenceInfo must** be a valid pointer to a valid **StdVideoDecodeH265ReferenceInfo** value

39.8. Video Encode Operations

Before the application can start recording Vulkan command buffers for the Video Encode Operations, it **must** do the following, beforehand:

1. Ensure that the implementation can encode the Video Content by querying the supported codec operations and profiles using **vkGetPhysicalDeviceQueueFamilyProperties2**.
2. By using **vkGetPhysicalDeviceVideoFormatPropertiesKHR** and providing one or more video profiles, choose the Vulkan formats supported by the implementation. The formats for **input**
and reference pictures must be queried and chosen separately. Refer to the section on enumeration of supported video formats.

3. Before creating an image to be used as a video picture resource, obtain the supported image creation parameters by querying with `vkGetPhysicalDeviceFormatProperties2` and `vkGetPhysicalDeviceImageFormatProperties2` using one of the reported formats and adding `VkVideoProfilesKHR` to the pNext chain of `VkFormatProperties2`. When querying the parameters with `vkGetPhysicalDeviceImageFormatProperties2` for images targeting input and reference (Dpb) pictures, the `VkPhysicalDeviceImageFormatInfo2::usage` field should contain `VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR` and `VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR`, respectively.

4. Create none, some, or all of the required images for the input and reference pictures. More Video Picture Resources can be created at some later point if needed while processing the content to be encoded. Also, if the size of the picture to be encoded is expected to change, the images can be created based on the maximum expected content size.

5. Create the video session to be used for video encode operations. Before creating the Encode Video Session, the encode capabilities should be queried with `vkGetPhysicalDeviceVideoCapabilitiesKHR` to obtain the limits of the parameters allowed by the implementation for a particular codec profile.

6. Bind memory resources with the encode video session by calling `vkBindVideoSessionMemoryKHR`. The video session cannot be used until memory resources are allocated and bound to it. In order to determine the required memory sizes and heap types of the device memory allocations, `vkGetVideoSessionMemoryRequirementsKHR` should be called.

7. Create one or more Session Parameter objects for use across command buffer recording operations, if required by the codec extension in use. These objects must be created against a video session with the parameters required by the codec. Each Session Parameter object created is a child object of the associated Session object and cannot be bound in the command buffer with any other Session Object.

The recording of Video Encode Commands against a Vulkan Command Buffer consists of the following sequence:

1. `vkCmdBeginVideoCodingKHR` starts the recording of one or more Video Encode operations in the command buffer. For each Video Encode Command operation, a Video Session must be bound to the command buffer within this command. This command establishes a Vulkan Video Encode Context that consists of the bound Video Session Object, Session Parameters Object, and the required Video Picture Resources. The established Video Encode Context is in effect until the `vkCmdEndVideoCodingKHR` command is recorded. If more Video Encode operations are to be required after the `vkCmdEndVideoCodingKHR` command, another Video Encode Context can be started with the `vkCmdBeginVideoCodingKHR` command.

2. `vkCmdEncodeVideoKHR` specifies one or more frames to be encoded. The `VkVideoEncodeInfoKHR` parameters, and the codec extension structures chained to this, specify the details of the encode operation.

3. `vkCmdControlVideoCodingKHR` records operations against the encoded data, encoding device, or the Video Session state.
4. `vkCmdEndVideoCodingKHR` signals the end of the recording of the Vulkan Video Encode Context, as established by `vkCmdBeginVideoCodingKHR`.

In addition to the above, the following commands can be recorded between `vkCmdBeginVideoCodingKHR` and `vkCmdEndVideoCodingKHR`:

- Query operations
- Global Memory Barriers
- Buffer Memory Barriers
- Image Memory Barriers (these must be used to transition the Video Picture Resources to the proper `VK_IMAGE_LAYOUT_VIDEO_ENCODE_SRC_KHR` and `VK_IMAGE_LAYOUT_VIDEO_ENCODE_DPB_KHR` layouts).
- Pipeline Barriers
- Events
- Timestamps
- Device Groups (device mask)

The following Video Encode related commands must be recorded outside the Vulkan Video Encode Context established with the `vkCmdBeginVideoCodingKHR` and `vkCmdEndVideoCodingKHR` commands:

- Sparse Memory Binding
- Copy Commands
- Clear Commands

39.8.1. Video Encode Vulkan Command Buffer Commands

To launch an encode operation that results in bitstream generation, call:

```c
// Provided by VK_KHR_video_encode_queue
void vkCmdEncodeVideoKHR(
    VkCommandBuffer                             commandBuffer, // Provided by VK_KHR_video_encode_queue
    const VkVideoEncodeInfoKHR*                 pEncodeInfo);
```

- `commandBuffer` is the command buffer to be filled with this function for encoding to generate a bitstream.
- `pEncodeInfo` is the pointer of `VkVideoEncodeInfoKHR`.  

Valid Usage (Implicit)

- **VUID-vkCmdEncodeVideoKHR-commandBuffer-parameter**
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- **VUID-vkCmdEncodeVideoKHR-pEncodeInfo-parameter**
  
  `pEncodeInfo` must be a valid pointer to a valid `VkVideoEncodeInfoKHR` structure

- **VUID-vkCmdEncodeVideoKHR-commandBuffer-recording**
  
  `commandBuffer` must be in the recording state

- **VUID-vkCmdEncodeVideoKHR-commandBuffer-cmdpool**
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support encode operations

- **VUID-vkCmdEncodeVideoKHR-renderpass**
  
  This command must only be called outside of a render pass instance

- **VUID-vkCmdEncodeVideoKHR-bufferlevel**
  
  `commandBuffer` must be a primary `VkCommandBuffer`

Host Synchronization

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Outside</td>
<td>Encode</td>
<td></td>
</tr>
</tbody>
</table>

The `VkVideoEncodeInfoKHR` structure is defined as:
typedef struct VkVideoEncodeInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkVideoEncodeFlagsKHR flags;
    uint32_t qualityLevel;
    VkExtent2D codedExtent;
    VkBuffer dstBitstreamBuffer;
    VkDeviceSize dstBitstreamBufferOffset;
    VkDeviceSize dstBitstreamBufferMaxRange;
    VkVideoPictureResourceKHR srcPictureResource;
    const VkVideoReferenceSlotKHR* pSetupReferenceSlot;
    uint32_t referenceSlotCount;
    const VkVideoReferenceSlotKHR* pReferenceSlots;
} VkVideoEncodeInfoKHR;

- **sType** is the type of this structure.
- **pNext** is a pointer to a structure extending this structure. A codec-specific extension structure must be chained to specify what bitstream unit to generate with this encode operation.
- **flags** is encode flags of type VkVideoEncodeFlagBitsKHR, reserved for future versions of this specification.
- **qualityLevel** is the coding quality level of the encoding. It is defined by the codec-specific extensions.
- **codedExtent** is the coded size of the encode operations.
- **dstBitstreamBuffer** is the buffer where the encoded bitstream output will be produced.
- **dstBitstreamBufferOffset** is the offset in the dstBitstreamBuffer where the encoded bitstream output will start. dstBitstreamBufferOffset's value must be aligned to VkVideoCapabilitiesKHR::minBitstreamBufferOffsetAlignment, as reported by the implementation.
- **dstBitstreamBufferMaxRange** is the maximum size of the dstBitstreamBuffer that can be used while the encoded bitstream output is produced. dstBitstreamBufferMaxRange's value must be aligned to VkVideoCapabilitiesKHR::minBitstreamBufferSizeAlignment, as reported by the implementation.
- **srcPictureResource** is the Picture Resource of the Input Picture to be encoded by the operation.
- **pSetupReferenceSlot** is an optional parameter used for generating a reconstructed reference slot and Picture Resource. VkVideoReferenceSlotKHR::slotIndex specifies the slot index number to use as a target for producing the Reconstructed (Dbp) data. pSetupReferenceSlot must be one of the entries provided in VkVideoBeginCodingInfoKHR via the pReferenceSlots within the vkCmdBeginVideoCodingKHR command that established the Vulkan Video Encode Context for this command.
- **referenceSlotCount** is the number of the Reconstructed Reference Pictures that will be used when this encoding operation is executing.
- **pReferenceSlots** is a list of the reference slots that will be used when this encoding operation is executing. Each entry in pReferenceSlots must be one of the entries provided in Chapter 39. Video Decode and Encode Operations | 2063
 VkVideoBeginCodingInfoKHR via the pReferenceSlots within the vkCmdBeginVideoCodingKHR command that established the Vulkan Video Encode Context for this command.

Multiple vkCmdEncodeVideoKHR commands may be recorded within a Vulkan Video Encode Context. The execution of each vkCmdEncodeVideoKHR command will result in generating codec-specific bitstream units. These bitstream units are generated consecutively into the bitstream buffer specified in dstBitstreamBuffer of VkVideoEncodeInfoKHR within the vkCmdBeginVideoCodingKHR command. The produced bitstream is the sum of all these bitstream units, including any padding between the bitstream units. Any bitstream padding must be filled with data compliant to the codec standard so as not to cause any syntax errors during decoding of the bitstream units with the padding included. The range of the bitstream buffer written can be queried via video encode bitstream buffer range queries.

Valid Usage (Implicit)

- VUID-VkVideoEncodeInfoKHR-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_ENCODE_INFO_KHR

- VUID-VkVideoEncodeInfoKHR-pNext-pNext
  Each pNext member of any structure (including this one) in the pNext chain must be either NULL or a pointer to a valid instance of VkVideoEncodeH264EmitPictureParametersEXT or VkVideoEncodeH264VclFrameInfoEXT

- VUID-VkVideoEncodeInfoKHR-sType-unique
  The sType value of each struct in the pNext chain must be unique

- VUID-VkVideoEncodeInfoKHR-flags-parameter
  flags must be a valid combination of VkVideoEncodeFlagBitsKHR values

- VUID-VkVideoEncodeInfoKHR-dstBitstreamBuffer-parameter
  dstBitstreamBuffer must be a valid VkBuffer handle

- VUID-VkVideoEncodeInfoKHR-srcPictureResource-parameter
  srcPictureResource must be a valid VkVideoPictureResourceKHR structure

- VUID-VkVideoEncodeInfoKHR-pSetupReferenceSlot-parameter
  pSetupReferenceSlot must be a valid pointer to a valid VkVideoReferenceSlotKHR structure

- VUID-VkVideoEncodeInfoKHR-pReferenceSlots-parameter
  pReferenceSlots must be a valid pointer to an array of referenceSlotCount valid VkVideoReferenceSlotKHR structures

- VUID-VkVideoEncodeInfoKHR-referenceSlotCount-arraylength
  referenceSlotCount must be greater than 0

The vkCmdEncodeVideoKHR flags are defined with the following enumeration:
typedef enum VkVideoEncodeFlagBitsKHR {
    VK_VIDEO_ENCODE_DEFAULT_KHR = 0,
    VK_VIDEO_ENCODE_RESERVED_0_BIT_KHR = 0x00000001,
} VkVideoEncodeFlagBitsKHR;

• VK_VIDEO_ENCODE_RESERVED_0_BIT_KHR The current version of the specification has reserved this value for future use.

typedef VkFlags VkVideoEncodeFlagsKHR;

VkVideoEncodeFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoEncodeFlagBitsKHR.

The VkVideoEncodeRateControlInfoKHR structure is defined as:

typedef struct VkVideoEncodeRateControlInfoKHR {
    VkStructureType sType;
    const void* pNext;
    VkVideoEncodeRateControlFlagsKHR flags;
    VkVideoEncodeRateControlModeFlagBitsKHR rateControlMode;
    uint32_t averageBitrate;
    uint16_t peakToAverageBitrateRatio;
    uint16_t frameRateNumerator;
    uint16_t frameRateDenominator;
    uint32_t virtualBufferSizeInMs;
} VkVideoEncodeRateControlInfoKHR;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• flags is encode rate control flags.
• rateControlMode is encode rate control mode from VkVideoEncodeRateControlModeFlagBitsKHR.
• averageBitrate is average bitrate in bits/second. Valid when rate control is not VK_VIDEO_ENCODE_RATE_CONTROL_MODE_NONE_BIT_KHR.
• peakToAverageBitrateRatio is peak bitrate to average bitrate in percentage. Valid when rate control is VK_VIDEO_ENCODE_RATE_CONTROL_MODE_VBR_BIT_KHR.
• frameRateNumerator is numerator of frame rate. Valid when rate control is not VK_VIDEO_ENCODE_RATE_CONTROL_MODE_NONE_BIT_KHR.
• frameRateDenominator is denominator of frame rate. Valid when rate control is not VK_VIDEO_ENCODE_RATE_CONTROL_MODE_NONE_BIT_KHR.
• virtualBufferSizeInMs is leaky bucket model virtual buffer size in milliseconds with respect to
peak bitrate. Valid when rate control is not VK_VIDEO_ENCODE_RATE_CONTROL_MODE_NONE_BIT_KHR. For example, virtual buffer size is \((\text{virtualBufferSizeInMs} \times \text{peakToAverageBitrateRatio} \times \text{averageBitrate} / 100000)\).

A codec-specific extension structure for further rate control parameter settings may be chained to VkVideoEncodeRateControlInfoKHR.

### Valid Usage (Implicit)

- **VUID-VkVideoEncodeRateControlInfoKHR-sType-sType**
  - `sType` must be VK_STRUCTURE_TYPE_VIDEO_ENCODE_RATE_CONTROL_INFO_KHR
- **VUID-VkVideoEncodeRateControlInfoKHR-flags-parameter**
  - `flags` must be a valid combination of VkVideoEncodeRateControlFlagBitsKHR values
- **VUID-VkVideoEncodeRateControlInfoKHR-flags-required bitmask**
  - `flags` must not be `0`
- **VUID-VkVideoEncodeRateControlInfoKHR-rateControlMode-parameter**
  - `rateControlMode` must be a valid VkVideoEncodeRateControlModeFlagBitsKHR value

The rate control modes are defined with the following enums:

```c
// Provided by VK_KHR_video_encode_queue
typedef enum VkVideoEncodeRateControlModeFlagBitsKHR {
    VK_VIDEO_ENCODE_RATE_CONTROL_MODE_NONE_BIT_KHR = 0,
    VK_VIDEO_ENCODE_RATE_CONTROL_MODE_CBR_BIT_KHR = 1,
    VK_VIDEO_ENCODE_RATE_CONTROL_MODE_VBR_BIT_KHR = 2,
} VkVideoEncodeRateControlModeFlagBitsKHR;
```

- **VK_VIDEO_ENCODE_RATE_CONTROL_MODE_NONE_BIT_KHR** for disabling rate control.
- **VK_VIDEO_ENCODE_RATE_CONTROL_MODE_CBR_BIT_KHR** for constant bitrate rate control mode.
- **VK_VIDEO_ENCODE_RATE_CONTROL_MODE_VBR_BIT_KHR** for variable bitrate rate control mode.

```c
// Provided by VK_KHR_video_encode_queue
typedef VkFlags VkVideoEncodeRateControlModeFlagsKHR;
```

VkVideoEncodeRateControlModeFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoEncodeRateControlModeFlagBitsKHR.

The `vkCmdControlVideoCodingKHR` flags are defined with the following enumeration:
typedef enum VkVideoEncodeRateControlFlagBitsKHR {
    VK_VIDEO_ENCODE_RATE_CONTROL_DEFAULT_KHR = 0,
    VK_VIDEO_ENCODE_RATE_CONTROL_RESET_BIT_KHR = 0x00000001,
} VkVideoEncodeRateControlFlagBitsKHR;

- VK_VIDEO_ENCODE_RESERVED_0_BIT_KHR The current version of the specification has reserved this value for future use.

typedef VkFlags VkVideoEncodeRateControlFlagsKHR;

VkVideoEncodeRateControlFlagsKHR is a bitmask type for setting a mask of zero or more VkVideoEncodeRateControlFlagBitsKHR.

### 39.9. Encode H.264

This extension adds H.264 codec specific structures/types needed to support H.264 encoding. Unless otherwise noted, all references to the H.264 specification are to the 2010 edition published by the ITU-T, dated March 2010. This specification is available at [http://www.itu.int/rec/T-REC-H.264](http://www.itu.int/rec/T-REC-H.264).

#### 39.9.1. H.264 encode profile

A H.264 encode profile is specified using VkVideoEncodeH264ProfileEXT chained to VkVideoProfileKHR when the codec-operation in VkVideoProfileKHR is VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_EXT.

The VkVideoEncodeH264ProfileEXT structure is defined as:

typedef struct VkVideoEncodeH264ProfileEXT {
    VkStructureType               sType;
    const void*                   pNext;
    StdVideoH264ProfileIdc         stdProfileIdc;
} VkVideoEncodeH264ProfileEXT;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- stdProfileIdc is the H.264 codec profile IDC defined in StdVideoH264ProfileIdc for enumeration.

**Valid Usage (Implicit)**

- VUID-VkVideoEncodeH264ProfileEXT-sType-sType
  
  sType **must be** VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_PROFILE_EXT
### 39.9.2. Capabilities

When `vkGetPhysicalDeviceVideoCapabilitiesKHR` is called to query the capabilities with parameter `videoCodecOperation` specified as `VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_EXT`, an instance of `VkVideoEncodeH264CapabilitiesEXT` structure can be chained to `VkVideoCapabilitiesKHR` to retrieve H.264 extension specific capabilities.

The `VkVideoEncodeH264CapabilitiesEXT` structure is defined as:

```c
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264CapabilitiesEXT {
    VkStructureType                          sType;
    const void*                              pNext;
    VkVideoEncodeH264CapabilitiesFlagsEXT    flags;
    VkVideoEncodeH264InputModeFlagsEXT       inputModeFlags;
    VkVideoEncodeH264OutputModeFlagsEXT      outputModeFlags;
    VkExtent2D                               minPictureSizeInMbs;
    VkExtent2D                               maxPictureSizeInMbs;
    VkExtent2D                               inputImageDataAlignment;
    uint8_t                                  maxNumL0ReferenceForP;
    uint8_t                                  maxNumL0ReferenceForB;
    uint8_t                                  maxNumL1Reference;
    uint8_t                                  qualityLevelCount;
    VkExtensionProperties                    stdExtensionVersion;
} VkVideoEncodeH264CapabilitiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` reports various encoding tools supported from those defined in `VkVideoEncodeH264CapabilitiesFlagBitsEXT`.
- `inputModeFlags` reports supported command buffer input granularities/modes from those defined in `VkVideoEncodeH264InputModeFlagBitsEXT`.
- `outputModeFlags` reports supported output (bitstream size reporting) granularities/modes from those defined in `VkVideoEncodeH264OutputModeFlagBitsEXT`.
- `maxPictureSizeInMbs` reports the maximum encoded picture width and height in Macroblocks.
- `minPictureSizeInMbs` reports the minimum encoded picture width and height in Macroblocks.
- `inputImageDataAlignment` reports alignment of data that **must** be filled in the input image horizontally and vertically in pixels. Max is 16: Data **must** be filled up to Macroblock boundary (no extrapolation/filling is supported by implementation). Min is 1: The implementation will take care of all needed extrapolation/filling up to the next Macroblock boundary.
- `maxNumL0ReferenceForP` reports the maximum number of reference pictures the encoder supports as list0 references for P pictures.
- `maxNumL0ReferenceForB` reports the maximum number of reference pictures the encoder supports as list0 references for B pictures. Reports 0 if B pictures are not supported.
• **maxNumL1Reference** reports the maximum number of reference pictures the encoder supports as list1 references if B pictures are supported. Reports 0 if B pictures are not supported.

• **qualityLevelCount** is the number of discrete quality levels supported.

• **stdExtensionVersion** is the specific H.264 extension name and version supported by this implementation.

---

**Valid Usage (Implicit)**

- VUID-VkVideoEncodeH264CapabilitiesEXT-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_CAPABILITIES_EXT**

- VUID-VkVideoEncodeH264CapabilitiesEXT-flags-parameter
  
  **flags** must be a valid combination of **VkVideoEncodeH264CapabilitiesFlagBitsEXT** values

- VUID-VkVideoEncodeH264CapabilitiesEXT-flags-requiredbitmask
  
  **flags** must not be 0

- VUID-VkVideoEncodeH264CapabilitiesEXT-inputModeFlags-parameter
  
  **inputModeFlags** must be a valid combination of **VkVideoEncodeH264InputModeFlagBitsEXT** values

- VUID-VkVideoEncodeH264CapabilitiesEXT-inputModeFlags-requiredbitmask
  
  **inputModeFlags** must not be 0

- VUID-VkVideoEncodeH264CapabilitiesEXT-outputModeFlags-parameter
  
  **outputModeFlags** must be a valid combination of **VkVideoEncodeH264OutputModeFlagBitsEXT** values

- VUID-VkVideoEncodeH264CapabilitiesEXT-outputModeFlags-requiredbitmask
  
  **outputModeFlags** must not be 0

- VUID-VkVideoEncodeH264CapabilitiesEXT-stdExtensionVersion-parameter
  
  **stdExtensionVersion** must be a valid **VkExtensionProperties** structure

---

The **flags** field reports various encoding tools supported as defined with the following bitfield:
// Provided by VK_EXT_video_encode_h264

typedef enum VkVideoEncodeH264CapabilitiesFlagBitsEXT {
    VK_VIDEO_ENCODE_H264_CAPABILITY_CABAC_BIT_EXT = 0x00000001,
    VK_VIDEO_ENCODE_H264_CAPABILITY_CAVLC_BIT_EXT = 0x00000002,
    VK_VIDEO_ENCODE_H264_CAPABILITY_WEIGHTED_BI_PRED_IMPLICIT_BIT_EXT = 0x00000004,
    VK_VIDEO_ENCODE_H264_CAPABILITY_TRANSFORM_8X8_BIT_EXT = 0x00000008,
    VK_VIDEO_ENCODE_H264_CAPABILITY_CHROMA_QP_OFFSET_BIT_EXT = 0x00000010,
    VK_VIDEO_ENCODE_H264_CAPABILITY_SECOND_CHROMA_QP_OFFSET_BIT_EXT = 0x00000020,
    VK_VIDEO_ENCODE_H264_CAPABILITY_DEBLOCKING_FILTER_DISABLED_BIT_EXT = 0x00000040,
    VK_VIDEO_ENCODE_H264_CAPABILITY_DEBLOCKING_FILTER_ENABLED_BIT_EXT = 0x00000080,
    VK_VIDEO_ENCODE_H264_CAPABILITY_DEBLOCKING_FILTER_PARTIAL_BIT_EXT = 0x00000100,
    VK_VIDEO_ENCODE_H264_CAPABILITY_MULTIPLE_SLICE_PER_FRAME_BIT_EXT = 0x00000200,
    VK_VIDEO_ENCODE_H264_CAPABILITY_EVENLY_DISTRIBUTED_SLICE_SIZE_BIT_EXT = 0x00000400,
} VkVideoEncodeH264CapabilitiesFlagBitsEXT;

- **VK_VIDEO_ENCODE_H264_CAPABILITY_CABAC_BIT_EXT** reports if CABAC entropy coding is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_CAVLC_BIT_EXT** reports if CAVLC entropy coding is supported. An implementation **must** support at least one entropy coding mode.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_WEIGHTED_BI_PRED_IMPLICIT_BIT_EXT** reports if using weighted_bipred_idc_flag from StdVideoH264PpsFlags is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_TRANSFORM_8X8_BIT_EXT** reports if enabling transform_8x8_mode_flag in StdVideoH264PpsFlags is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_CHROMA_QP_OFFSET_BIT_EXT** reports if setting chroma_qp_index_offset in StdVideoH264PictureParameterSet is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_SECOND_CHROMA_QP_OFFSET_BIT_EXT** reports if setting second_chroma_qp_index_offset in StdVideoH264PictureParameterSet is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_DEBLOCKING_FILTER_DISABLED_BIT_EXT** reports if using std_video_h264_disable_deblocking_filter_idc_disabled from StdVideoH264DisableDeblockingFilterIdc is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_DEBLOCKING_FILTER_ENABLED_BIT_EXT** reports if using std_video_h264_disable_deblocking_filter_idc_enabled from StdVideoH264DisableDeblockingFilterIdc is supported.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_DEBLOCKING_FILTER_PARTIAL_BIT_EXT** reports if using std_video_h264_disable_deblocking_filter_idc_partial from StdVideoH264DisableDeblockingFilterIdc is supported. An implementation **must** support at least one deblocking filter mode.
- **VK_VIDEO_ENCODE_H264_CAPABILITY_MULTIPLE_SLICE_PER_FRAME_BIT_EXT**
- **VK_VIDEO_ENCODE_H264_CAPABILITY_EVENLY_DISTRIBUTED_SLICE_SIZE_BIT_EXT**

The **inputModeFlags** field reports the various command buffer input granularities supported by the implementation as follows:
typedef enum VkVideoEncodeH264InputModeFlagBitsEXT {
    VK_VIDEO_ENCODE_H264_INPUT_MODE_FRAME_BIT_EXT = 0x00000001,
    VK_VIDEO_ENCODE_H264_INPUT_MODE_SLICE_BIT_EXT = 0x00000002,
    VK_VIDEO_ENCODE_H264_INPUT_MODE_NON_VCL_BIT_EXT = 0x00000004,
} VkVideoEncodeH264InputModeFlagBitsEXT;

• VK_VIDEO_ENCODE_H264_INPUT_MODE_FRAME_BIT_EXT indicates a single command buffer must at least encode an entire frame. Any non-VCL NALUs must be encoded using the same command buffer as the frame if VK_VIDEO_ENCODE_H264_INPUT_MODE_NON_VCL_BIT_EXT is not supported.

• VK_VIDEO_ENCODE_H264_INPUT_MODE_SLICE_BIT_EXT indicates a single command buffer must at least encode a single slice. Any non-VCL NALUs must be encoded using the same command buffer as the first slice of the frame if VK_VIDEO_ENCODE_H264_INPUT_MODE_NON_VCL_BIT_EXT is not supported.

• VK_VIDEO.Encode.H264_INPUT_MODE_NON_VCL_BIT_EXT indicates a single command buffer may encode a non-VCL NALU by itself.

An implementation must support at least one of VK_VIDEO_ENCODE_H264_INPUT_MODE_FRAME_BIT_EXT or VK_VIDEO_ENCODE_H264_INPUT_MODE_SLICE_BIT_EXT.

The outputModeFlags field reports the minimum bitstream generation commands that must be included between each vkCmdBeginVideoCodingKHR and vkCmdEncodeVideoKHR pair (henceforth simply begin/end pair). The various output modes are:

typedef enum VkVideoEncodeH264OutputModeFlagBitsEXT {
    VK_VIDEO_ENCODE_H264_OUTPUT_MODE_FRAME_BIT_EXT = 0x00000001,
    VK_VIDEO_ENCODE_H264_OUTPUT_MODE_SLICE_BIT_EXT = 0x00000002,
    VK_VIDEO_ENCODE_H264_OUTPUT_MODE_NON_VCL_BIT_EXT = 0x00000004,
} VkVideoEncodeH264OutputModeFlagBitsEXT;

• VK_VIDEO_ENCODE_H264_OUTPUT_MODE_FRAME_BIT_EXT indicates calls to generate all NALUs of a frame must be included within a single begin/end pair. Any non-VCL NALUs must be encoded within the same begin/end pair if VK_VIDEO_ENCODE_H264_OUTPUT_MODE_NON_VCL_BIT_EXT is not supported.

• VK_VIDEO_ENCODE_H264_OUTPUT_MODE_SLICE_BIT_EXT indicates each begin/end pair must encode at least one slice. Any non-VCL NALUs must be encoded within the same begin/end pair as the first slice of the frame if VK_VIDEO_ENCODE_H264_OUTPUT_MODE_NON_VCL_BIT_EXT is not supported.

• VK_VIDEO_ENCODE_H264_OUTPUT_MODE_NON_VCL_BIT_EXT indicates each begin/end pair may encode only a non-VCL NALU by itself. An implementation must support at least one of VK_VIDEO_ENCODE_H264_OUTPUT_MODE_FRAME_BIT_EXT or VK_VIDEO_ENCODE_H264_OUTPUT_MODE_SLICE_BIT_EXT.

A single begin/end pair must not encode more than a single frame.

The bitstreams of NALUs generated within a single begin/end pair are written continuously into the same bitstream buffer (any padding between the NALUs must be compliant to the H.264 standard).
The supported input modes must be coarser or equal to the supported output modes. For example, it is illegal to report slice input is supported but only frame output is supported.

An implementation must report one of the following combinations of input/output modes:
- **Input:** Frame, **Output:** Frame
- **Input:** Frame, **Output:** Slice
- **Input:** Slice, **Output:** Slice
- **Input:** Frame and Non-VCL, **Output:** Frame and Non-VCL
- **Input:** Slice and Non-VCL, **Output:** Slice and Non-VCL

**39.9.3. Create Information**

An instance of `VkVideoEncodeH264SessionCreateInfoEXT` structure must be chained to `VkVideoSessionCreateInfoKHR` when the function `vkCreateVideoSessionKHR` is called with `videoCodecOperation` in `VkVideoSessionCreateInfoKHR` set to `VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_EXT`.

The `VkVideoEncodeH264SessionCreateInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264SessionCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkVideoEncodeH264CreateFlagsEXT flags;
    VkExtent2D maxPictureSizeInMbs;
    const VkExtensionProperties* pStdExtensionVersion;
} VkVideoEncodeH264SessionCreateInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `flags` is H.264 encoder creation flags.
- `maxPictureSizeInMbs` specifies the syntax element `pic_width_in_mbs_minus1 + 1` and the syntax element `pic_height_in_map_units_minus1 + 1`.
- `pStdExtensionVersion` is the specific H.264 codec extensions defined in `StdVideoH264Extensions`.

**Valid Usage (Implicit)**

- VUID-VkVideoEncodeH264SessionCreateInfoEXT-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_CREATE_INFO_EXT`
- VUID-VkVideoEncodeH264SessionCreateInfoEXT-flags-parameter
  - `flags` must be a valid combination of `VkVideoEncodeH264CreateFlagBitsEXT` values
- VUID-VkVideoEncodeH264SessionCreateInfoEXT-flags-requiredbitmask
  - `flags` must not be `0`
- VUID-VkVideoEncodeH264SessionCreateInfoEXT-pStdExtensionVersion-parameter
  - `pStdExtensionVersion` must be a valid pointer to a valid `VkExtensionProperties` structure
39.9.4. Encoder Parameter Sets

To reduce parameter traffic during encoding, the encoder parameter set object supports storing H.264 SPS/PPS parameter sets that may be later referenced during encoding.

An instance of VkVideoEncodeH264SessionParametersCreateInfoEXT holding one H.264 SPS and at least one H.264 PPS parameter set must be chained to VkVideoSessionParametersCreateInfoKHR when calling vkCreateVideoSessionParametersKHR to store these parameter set(s) with the encoder parameter set object for later reference. The provided H.264 SPS/PPS parameters must be within the limits specified during encoder creation for the encoder specified in VkVideoSessionParametersCreateInfoKHR.

The VkVideoEncodeH264SessionParametersCreateInfoEXT structure is defined as:

```c
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264SessionParametersCreateInfoEXT {
    VkStructureType sType;  
    const void* pNext;     
    uint32_t maxSpsStdCount;  
    uint32_t maxPpsStdCount;  
    const VkVideoEncodeH264SessionParametersAddInfoEXT* pParametersAddInfo;  
} VkVideoEncodeH264SessionParametersCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxSpsStdCount** is the maximum number of SPS parameters that the VkVideoSessionParametersKHR can contain.
- **maxPpsStdCount** is the maximum number of PPS parameters that the VkVideoSessionParametersKHR can contain.
- **pParametersAddInfo** is a pointer to an optional set of H.264 parameters to add upon object creation.

**Valid Usage (Implicit)**

- VUID-VkVideoEncodeH264SessionParametersCreateInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_PARAMETERS_CREATE_INFO_EXT

- VUID-VkVideoEncodeH264SessionParametersCreateInfoEXT-pParametersAddInfo-parameter
  If pParametersAddInfo is not NULL, pParametersAddInfo must be a valid pointer to a valid VkVideoEncodeH264SessionParametersAddInfoEXT structure
typedef struct VkVideoEncodeH264SessionParametersAddInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint32_t spsStdCount;
    const StdVideoH264SequenceParameterSet* pSpsStd;
    uint32_t ppsStdCount;
    const StdVideoH264PictureParameterSet* pPpsStd;
} VkVideoEncodeH264SessionParametersAddInfoEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **spsStdCount** is the number of SPS elements in the `pSpsStd`. Its value **must** be less than or equal to the value of `maxSpsStdCount`.
- **pSpsStd** is an array of structures representing H.264 sequence parameter sets. Each `StdVideoH264SequenceParameterSet` entry **must** have a unique H.264 SPS ID.
- **ppsStdCount** is the number of PPS provided in `pPpsStd`. Its value **must** be less than or equal to the value of `maxPpsStdCount`.
- **pPpsStd** is an array of structures representing H.264 picture parameter sets. Each `StdVideoH264PictureParameterSet` entry **must** have a unique H.264 SPS-PPS ID pair.
Valid Usage

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-spsStdCount-04837**
  The values of `spsStdCount` and `ppsStdCount` must be less than or equal to the values of `maxSpsStdCount` and `maxPpsStdCount`, respectively.

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-maxSpsStdCount-04838**
  When the `maxSpsStdCount` number of parameters of type `StdVideoH264SequenceParameterSet` in the Video Session Parameters object is reached, no additional parameters of that type can be added to the object. `VK_ERROR_TOO_MANY_OBJECTS` will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-maxPpsStdCount-04839**
  When the `maxPpsStdCount` number of parameters of type `StdVideoH264PictureParameterSet` in the Video Session Parameters object is reached, no additional parameters of that type can be added to the object. `VK_ERROR_TOO_MANY_OBJECTS` will be returned if an attempt is made to add additional data to this object at this point.

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-None-04840**
  Each entry to be added must have a unique, to the rest of the parameter array entries and the existing parameters in the Video Session Parameters Object that is being updated, SPS-PPS IDs.

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-None-04841**
  Parameter entries that already exist in Video Session Parameters object with a particular SPS-PPS IDs cannot be replaced nor updated.

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-None-04842**
  When creating a new object using a Video Session Parameters as a template, the array's parameters with the same SPS-PPS IDs as the ones from the template take precedence.

- **VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-None-04843**
  SPS/PPS parameters must comply with the limits specified in `VkVideoSessionCreateInfoKHR` during Video Session creation.
Valid Usage (Implicit)

- VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_PARAMETERS_ADD_INFO_EXT

- VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-pSpsStd-parameter
  If pSpsStd is not NULL, pSpsStd must be a valid pointer to an array of spsStdCount StdVideoH264SequenceParameterSet values

- VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-pPpsStd-parameter
  If pPpsStd is not NULL, pPpsStd must be a valid pointer to an array of ppsStdCount StdVideoH264PictureParameterSet values

- VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-spsStdCount-arraylength
  spsStdCount must be greater than 0

- VUID-VkVideoEncodeH264SessionParametersAddInfoEXT-ppsStdCount-arraylength
  ppsStdCount must be greater than 0

39.9.5. Frame Encoding

To encode a frame the structure VkVideoEncodeH264VclFrameInfoEXT may be chained to VkVideoEncodeInfoKHR when calling vkCmdEncodeVideoKHR.

The structure VkVideoEncodeH264VclFrameInfoEXT representing a frame encode operation is defined as:

```
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264VclFrameInfoEXT {
    VkStructureType sType;
    const void* pNext;
    uint8_t refDefaultFinalList0EntryCount;
    const VkVideoEncodeH264DpbSlotInfoEXT* pRefDefaultFinalList0Entries;
    uint8_t refDefaultFinalList1EntryCount;
    const VkVideoEncodeH264DpbSlotInfoEXT* pRefDefaultFinalList1Entries;
    uint32_t naluSliceEntryCount;
    const VkVideoEncodeH264NaluSliceEXT* pNaluSliceEntries;
    const VkVideoEncodeH264DpbSlotInfoEXT* pCurrentPictureInfo;
} VkVideoEncodeH264VclFrameInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- refDefaultFinalList0EntryCount is the default number of reference List0 entries for all slices of this frame. Each slice may override this by providing its own List0 entries.
- pRefDefaultFinalList0Entries is an array of VkVideoEncodeH264DpbSlotInfoEXT providing info for each reference List0 entry. The entries provided must be ordered after all reference List0 modification operations are applied (i.e. final list order). The entries provided must not reflect decoded picture marking operations in this frame that are applicable to references; the impact...
of such operations must be reflected in future frame encode cmds. The slot index in each entry must match one of the slot indexes provided in pReferenceSlots of VkVideoEncodeInfoKHR structure to which this structure is chained to.

- refDefaultFinalList1EntryCount is the default number of reference List1 entries for all slices of this frame. Each slice may override this by providing its own List1 entries.

- pRefDefaultFinalList1Entries is an array of VkVideoEncodeH264DpbSlotInfoEXT providing info for each reference List1 entry. The entries provided must be ordered after all reference List1 modification operations are applied (i.e. final list order). The entries provided must not reflect decoded picture marking operations in this frame that are applicable to references; the impact of such operations must be reflected in future frame encode cmds. The slot index in each entry must match one of the slot indexes provided in pReferenceSlots of VkVideoEncodeInfoKHR structure to which this structure is chained to.

- naluSliceEntryCount is the number of NALUs in this frame.

- pNaluSliceEntries is an array of naluCount VkVideoEncodeH264Nalu* structs that make up the frame. This is an ordered sequence; the NALUs are generated consecutively in the bitstream buffer (provided in bitstreamBuffer in VkVideoBeginCodingInfoKHR) in the same order as in this array.

- pCurrentPictureInfo is the VkVideoEncodeH264DpbSlotInfoEXT info for the reconstructed picture for this frame. Info provided must reflect decoded picture marking operations in this frame that are applicable to this frame. The slot index in pCurrentPictureInfo must match the slot index provided in pSetupReferenceSlot of VkVideoEncodeInfoKHR structure to which this structure is chained to.
Valid Usage (Implicit)

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-sType-sType**
  
  *sType* must be **VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_VCL_FRAME_INFO_EXT**

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-pRefDefaultFinalList0Entries-parameter**
  
  *pRefDefaultFinalList0Entries* must be a valid pointer to an array of *refDefaultFinalList0EntryCount* valid **VkVideoEncodeH264DpbSlotInfoEXT** structures

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-pRefDefaultFinalList1Entries-parameter**
  
  *pRefDefaultFinalList1Entries* must be a valid pointer to an array of *refDefaultFinalList1EntryCount* valid **VkVideoEncodeH264DpbSlotInfoEXT** structures

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-pNaluSliceEntries-parameter**
  
  *pNaluSliceEntries* must be a valid pointer to an array of *naluSliceEntryCount* valid **VkVideoEncodeH264NaluSliceEXT** structures

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-pCurrentPictureInfo-parameter**
  
  *pCurrentPictureInfo* must be a valid pointer to a valid **VkVideoEncodeH264DpbSlotInfoEXT** structure

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-refDefaultFinalList0EntryCount-arraylength**
  
  *refDefaultFinalList0EntryCount* must be greater than 0

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-refDefaultFinalList1EntryCount-arraylength**
  
  *refDefaultFinalList1EntryCount* must be greater than 0

- **VUID-VkVideoEncodeH264VclFrameInfoEXT-naluSliceEntryCount-arraylength**
  
  *naluSliceEntryCount* must be greater than 0

The **VkVideoEncodeH264DpbSlotInfoEXT** structure correlates a DPB slot index with codec-specific information and is defined as:

```c
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264DpbSlotInfoEXT {
    VkStructureType                     sType;
    const void*                         pNext;
    int8_t                               slotIndex;
    const StdVideoEncodeH264PictureInfo* pStdPictureInfo;
} VkVideoEncodeH264DpbSlotInfoEXT;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to a structure extending this structure.
- *slotIndex* is the slot index for this picture. *slotIndex* must match the *slotIndex* in pSetupReferenceSlot of **VkVideoEncodeInfoKHR** in the command used to encode the corresponding picture.
- *pStdPictureInfo* is the H.264 codec-spec defined syntax/info associated with this picture.
Valid Usage (Implicit)

- **VUID-VkVideoEncodeH264DpbSlotInfoEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_DPB_SLOT_INFO_EXT`

- **VUID-VkVideoEncodeH264DpbSlotInfoEXT-pNext-pNext**
  - `pNext` must be `NULL`

- **VUID-VkVideoEncodeH264DpbSlotInfoEXT-pStdPictureInfo-parameter**
  - `pStdPictureInfo` must be a valid pointer to a valid `StdVideoEncodeH264PictureInfo` value

The structure `VkVideoEncodeH264NaluSliceEXT` representing a slice is defined as:

```c
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264NaluSliceEXT {
    VkStructureType             sType;
    const void*                 pNext;
    const StdVideoEncodeH264SliceHeader* pSliceHeaderStd;
    uint32_t                     mbCount;
    uint8_t                      refFinalList0EntryCount;
    const VkVideoEncodeH264DpbSlotInfoEXT* pRefFinalList0Entries;
    uint8_t                      refFinalList1EntryCount;
    const VkVideoEncodeH264DpbSlotInfoEXT* pRefFinalList1Entries;
    uint32_t                     precedingNaluBytes;
    uint8_t                      minQp;
    uint8_t                      maxQp;
} VkVideoEncodeH264NaluSliceEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pSliceHeaderStd` is the slice header H.264 standard structure.
- `mbCount` is the number of macroblocks in this slice.
- `refFinalList0EntryCount` is the number of active references in List0. If zero, the default frame List0 provided in the enclosing `VkVideoEncodeH264VclFrameInfoEXT` is used. If non-zero, it overrides the default frame List0 for this slice.
- `pRefFinalList0Entries` is the DPB info for the active List0 references. The same restrictions related to order of entries and decoded picture marking operations described for List0 in `VkVideoEncodeH264VclFrameInfoEXT` apply here.
- `refFinalList1EntryCount` is the number of active references in List1. If zero, the default frame List1 provided in the enclosing `VkVideoEncodeH264VclFrameInfoEXT` is used. If non-zero, it overrides the default frame List1 for this slice.
- `pRefFinalList1Entries` is the DPB info for the active List1 references. The same restrictions related to order of entries and decoded picture marking operations described for List1 in `VkVideoEncodeH264VclFrameInfoEXT` apply here.
- `precedingNaluBytes` specifies the number of bytes consumed by non-VCL NALUs inserted before
this slice (not through this API). TODO: move to Frame/SliceRC struct?

- **minQp** is the minimum quantization parameter when rate control is enabled. TODO: move to Frame/SliceRC struct?

- **maxQp** is the maximum quantization parameter when rate control is enabled. TODO: move to Frame/SliceRC struct? When rate control is disabled, **minQp and maxQp must** be set to the same QP value.

### Valid Usage (Implicit)

- **VUID-VkVideoEncodeH264NaluSliceEXT-sType-sType**
  sType must be VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_NALU_SLICE_EXT

- **VUID-VkVideoEncodeH264NaluSliceEXT-pNext-pNext**
  pNext must be NULL

- **VUID-VkVideoEncodeH264NaluSliceEXT-pSliceHeaderStd-parameter**
  pSliceHeaderStd must be a valid pointer to a valid StdVideoEncodeH264SliceHeader value

- **VUID-VkVideoEncodeH264NaluSliceEXT-pRefFinalList0Entries-parameter**
  pRefFinalList0Entries must be a valid pointer to an array of refFinalList0EntryCount valid VkVideoEncodeH264DpbSlotInfoEXT structures

- **VUID-VkVideoEncodeH264NaluSliceEXT-pRefFinalList1Entries-parameter**
  pRefFinalList1Entries must be a valid pointer to an array of refFinalList1EntryCount valid VkVideoEncodeH264DpbSlotInfoEXT structures

- **VUID-VkVideoEncodeH264NaluSliceEXT-refFinalList0EntryCount-arraylength**
  refFinalList0EntryCount must be greater than 0

- **VUID-VkVideoEncodeH264NaluSliceEXT-refFinalList1EntryCount-arraylength**
  refFinalList1EntryCount must be greater than 0

The structure **VkVideoEncodeH264EmitPictureParametersEXT** is defined as:

```c
// Provided by VK_EXT_video_encode_h264
typedef struct VkVideoEncodeH264EmitPictureParametersEXT {
    VkStructureType    sType;
    const void*        pNext;
    uint8_t            spsId;
    VkBool32           emitSpsEnable;
    uint32_t           ppsIdEntryCount;
    const uint8_t*     ppsIdEntries;
} VkVideoEncodeH264EmitPictureParametersEXT;
```

- **sType** is the type of this structure.

- **pNext** is NULL or a pointer to a structure extending this structure.

- **spsId** is the H.264 SPS ID for the H.264 SPS to insert in the bitstream. The SPS ID must match the SPS provided in **spsStd** of **VkVideoEncodeH264SessionParametersCreateInfoEXT**. This is retrieved from the **VkVideoSessionParametersKHR** object provided in
 VkVideoBeginCodingInfoKHR.

- **emitSpsEnable** enables the emitting of the SPS structure with id of `spsId`.
- **ppsIdEntryCount** is the number of entries in the `ppsIdEntries`. If this parameter is "0" then no pps entries are going to be emitted in the bitstream.
- **ppsIdEntries** is the H.264 PPS IDs for the H.264 PPS to insert in the bitstream. The PPS IDs **must** match one of the IDs of the PPS(s) provided in `pPpsStd` of `VkVideoEncodeH264SessionParametersCreateInfoEXT` to identify the PPS parameter set to insert in the bitstream. This is retrieved from the `VkVideoSessionParametersKHR` object provided in `VkVideoBeginCodingInfoKHR`.

### Valid Usage (Implicit)

- **VUID-VkVideoEncodeH264EmitPictureParametersEXT-sType-sType**
  
  - **sType** must be `VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_EMIT_PICTURE_PARAMETERS_EXT`

- **VUID-VkVideoEncodeH264EmitPictureParametersEXT-ppsIdEntries-parameter**
  
  - `ppsIdEntries` must be a valid pointer to an array of `ppsIdEntryCount` `uint8_t` values

- **VUID-VkVideoEncodeH264EmitPictureParametersEXT-ppsIdEntryCount-arraylength**
  
  - `ppsIdEntryCount` must be greater than `0`
Chapter 40. Extending Vulkan

New functionality may be added to Vulkan via either new extensions or new versions of the core, or new versions of an extension in some cases.

This chapter describes how Vulkan is versioned, how compatibility is affected between different versions, and compatibility rules that are followed by the Vulkan Working Group.

40.1. Instance and Device Functionality

Commands that enumerate instance properties, or that accept a `VkInstance` object as a parameter, are considered instance-level functionality. Commands that enumerate physical device properties, or that accept a `VkDevice` object or any of a device's child objects as a parameter, are considered device-level functionality.

Note
Applications usually interface to Vulkan using a loader that implements only instance-level functionality, passing device-level functionality to implementations of the full Vulkan API on the system. In some circumstances, as these may be implemented independently, it is possible that the loader and device implementations on a given installation will support different versions. To allow for this and call out when it happens, the Vulkan specification enumerates device and instance level functionality separately - they have independent version queries.

Note
Vulkan 1.0 initially specified new physical device enumeration functionality as instance-level, requiring it to be included in an instance extension. As the capabilities of device-level functionality require discovery via physical device enumeration, this led to the situation where many device extensions required an instance extension as well. To alleviate this extra work, `VK_KHR_get_physical_device_properties2` (and subsequently Vulkan 1.1) redefined device-level functionality to include physical device enumeration.

40.2. Core Versions

The Vulkan Specification is regularly updated with bug fixes and clarifications. Occasionally new functionality is added to the core and at some point it is expected that there will be a desire to perform a large, breaking change to the API. In order to indicate to developers how and when these changes are made to the specification, and to provide a way to identify each set of changes, the Vulkan API maintains a version number.

40.2.1. Version Numbers

The Vulkan version number comprises four parts indicating the variant, major, minor and patch version of the Vulkan API Specification.
The **variant** indicates the variant of the Vulkan API supported by the implementation. This is always 0 for the Vulkan API.

*Note*
A non-zero variant indicates the API is a variant of the Vulkan API and applications will typically need to be modified to run against it. The variant field was a later addition to the version number, added in version 1.2.175 of the Specification. As Vulkan uses variant 0, this change is fully backwards compatible with the previous version number format for Vulkan implementations. New version number macros have been added for this change and the old macros deprecated. For existing applications using the older format and macros, an implementation with non-zero variant will decode as a very high Vulkan version. The high version number should be detectable by applications performing suitable version checking.

The **major version** indicates a significant change in the API, which will encompass a wholly new version of the specification.

The **minor version** indicates the incorporation of new functionality into the core specification.

The **patch version** indicates bug fixes, clarifications, and language improvements have been incorporated into the specification.

Compatibility guarantees made about versions of the API sharing any of the same version numbers are documented in [Core Versions](#).

The version number is used in several places in the API. In each such use, the version numbers are packed into a 32-bit integer as follows:

- The variant is a 3-bit integer packed into bits 31-29.
- The major version is a 7-bit integer packed into bits 28-22.
- The minor version number is a 10-bit integer packed into bits 21-12.
- The patch version number is a 12-bit integer packed into bits 11-0.

`VK_API_VERSION_VARIANT` extracts the API variant number from a packed version number:

```c
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_VARIANT(version) ((uint32_t)(version) >> 29)
```

`VK_API_VERSION_MAJOR` extracts the API major version number from a packed version number:

```c
// Provided by VK_VERSION_1_0
#define VK_API_VERSION_MAJOR(version) (((uint32_t)(version) >> 22) & 0x7FU)
```

`VK_VERSION_MAJOR` extracts the API major version number from a packed version number:
```c
// Provided by VK_VERSION_1_0
// DEPRECATED: This define is deprecated. VK_API_VERSION_MAJOR should be used instead.
#define VK_VERSION_MAJOR(version) ((uint32_t)(version) >> 22)

VK_API_VERSION_MINOR extracts the API minor version number from a packed version number:

// Provided by VK_VERSION_1_0
#define VK_API_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x3FFU)

VK_VERSION_MINOR extracts the API minor version number from a packed version number:

// Provided by VK_VERSION_1_0
#define VK_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x3FFU)

VK_API_VERSION_PATCH extracts the API patch version number from a packed version number:

// Provided by VK_VERSION_1_0
#define VK_API_VERSION_PATCH(version) ((uint32_t)(version) & 0xFFFU)

VK_VERSION_PATCH extracts the API patch version number from a packed version number:

// Provided by VK_VERSION_1_0
#define VK_VERSION_PATCH(version) ((uint32_t)(version) & 0xFFFU)

VK_MAKE_API_VERSION constructs an API version number.

// Provided by VK_VERSION_1_0
#define VK_MAKE_API_VERSION(variant, major, minor, patch) \\
  (((uint32_t)(variant)) << 29) | (((uint32_t)(major)) << 22) | (((uint32_t)(minor)) << 12) | ((uint32_t)(patch))
```

- `variant` is the variant number.
- `major` is the major version number.
- `minor` is the minor version number.
- `patch` is the patch version number.

```c
VK_MAKE_VERSION constructs an API version number.
```
• **major** is the major version number.
• **minor** is the minor version number.
• **patch** is the patch version number.

**VK_API_VERSION_1_0** returns the API version number for Vulkan 1.0.0.

**VK_API_VERSION_1_1** returns the API version number for Vulkan 1.1.0.

### 40.2.2. Querying Version Support

The version of instance-level functionality can be queried by calling `vkEnumerateInstanceVersion`.

The version of device-level functionality can be queried by calling `vkGetPhysicalDeviceProperties` or `vkGetPhysicalDeviceProperties2`, and is returned in `VkPhysicalDeviceProperties::apiVersion`, encoded as described in [Version Numbers](#).

### 40.3. Layers

When a layer is enabled, it inserts itself into the call chain for Vulkan commands the layer is interested in. Layers can be used for a variety of tasks that extend the base behavior of Vulkan beyond what is required by the specification - such as call logging, tracing, validation, or providing additional extensions.

**Note**

For example, an implementation is not expected to check that the value of enums used by the application fall within allowed ranges. Instead, a validation layer would do those checks and flag issues. This avoids a performance penalty during production use of the application because those layers would not be enabled in production.
Note

Vulkan layers **may** wrap object handles (i.e. return a different handle value to the application than that generated by the implementation). This is generally discouraged, as it increases the probability of incompatibilities with new extensions. The validation layers wrap handles in order to track the proper use and destruction of each object. See the “Vulkan Loader Specification and Architecture Overview” document for additional information.

To query the available layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateInstanceLayerProperties(
    uint32_t* pPropertyCount,
    VkLayerProperties* pProperties);
```

- **pPropertyCount** is a pointer to an integer related to the number of layer properties available or queried, as described below.
- **pProperties** is either **NULL** or a pointer to an array of **VkLayerProperties** structures.

If **pProperties** is **NULL**, then the number of layer properties available is returned in **pPropertyCount**. Otherwise, **pPropertyCount** must point to a variable set by the user to the number of elements in the **pProperties** array, and on return the variable is overwritten with the number of structures actually written to **pProperties**. If **pPropertyCount** is less than the number of layer properties available, at most **pPropertyCount** structures will be written, and **VK_INCOMPLETE** will be returned instead of **VK_SUCCESS**, to indicate that not all the available properties were returned.

The list of available layers may change at any time due to actions outside of the Vulkan implementation, so two calls to **vkEnumerateInstanceLayerProperties** with the same parameters **may** return different results, or retrieve different **pPropertyCount** values or **pProperties** contents. Once an instance has been created, the layers enabled for that instance will continue to be enabled and valid for the lifetime of that instance, even if some of them become unavailable for future instances.

**Valid Usage (Implicit)**

- **VUID-vkEnumerateInstanceLayerProperties-pPropertyCount-parameter**
  
  **pPropertyCount** must be a valid pointer to a **uint32_t** value

- **VUID-vkEnumerateInstanceLayerProperties-pProperties-parameter**
  
  If the value referenced by **pPropertyCount** is not **0**, and **pProperties** is not **NULL**, **pProperties** must be a valid pointer to an array of **pPropertyCount** **VkLayerProperties** structures
Return Codes

Success
- VK_SUCCESS
- VK_INCOMPLETE

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OFDEVICE_MEMORY

The `VkLayerProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkLayerProperties {
    char layerName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
    uint32_t implementationVersion;
    char description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

- `layerName` is an array of `VK_MAX_EXTENSION_NAME_SIZE` `char` containing a null-terminated UTF-8 string which is the name of the layer. Use this name in the `ppEnabledLayerNames` array passed in the `VkInstanceCreateInfo` structure to enable this layer for an instance.
- `specVersion` is the Vulkan version the layer was written to, encoded as described in Version Numbers.
- `implementationVersion` is the version of this layer. It is an integer, increasing with backward compatible changes.
- `description` is an array of `VK_MAX_DESCRIPTION_SIZE` `char` containing a null-terminated UTF-8 string which provides additional details that can be used by the application to identify the layer.

To enable a layer, the name of the layer should be added to the `ppEnabledLayerNames` member of `VkInstanceCreateInfo` when creating a `VkInstance`.

Loader implementations may provide mechanisms outside the Vulkan API for enabling specific layers. Layers enabled through such a mechanism are implicitly enabled, while layers enabled by including the layer name in the `ppEnabledLayerNames` member of `VkInstanceCreateInfo` are explicitly enabled. Implicitly enabled layers are loaded before explicitly enabled layers, such that implicitly enabled layers are closer to the application, and explicitly enabled layers are closer to the driver. Except where otherwise specified, implicitly enabled and explicitly enabled layers differ only in the way they are enabled, and the order in which they are loaded. Explicitly enabling a layer that is implicitly enabled results in this layer being loaded as an implicitly enabled layer; it has no additional effect.
40.3.1. Device Layer Deprecation

Previous versions of this specification distinguished between instance and device layers. Instance layers were only able to intercept commands that operate on `VkInstance` and `VkPhysicalDevice`, except they were not able to intercept `vkCreateDevice`. Device layers were enabled for individual devices when they were created, and could only intercept commands operating on that device or its child objects.

Device-only layers are now deprecated, and this specification no longer distinguishes between instance and device layers. Layers are enabled during instance creation, and are able to intercept all commands operating on that instance or any of its child objects. At the time of deprecation there were no known device-only layers and no compelling reason to create one.

In order to maintain compatibility with implementations released prior to device-layer deprecation, applications **should** still enumerate and enable device layers. The behavior of `vkEnumerateDeviceLayerProperties` and valid usage of the `ppEnabledLayerNames` member of `VkDeviceCreateInfo` maximizes compatibility with applications written to work with the previous requirements.

To enumerate device layers, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceLayerProperties(
    VkPhysicalDevice                            physicalDevice,  
    uint32_t*                                   pPropertyCount,  
    VkLayerProperties*                          pProperties);
```

- `pPropertyCount` is a pointer to an integer related to the number of layer properties available or queried.
- `pProperties` is either `NULL` or a pointer to an array of `VkLayerProperties` structures.

If `pProperties` is `NULL`, then the number of layer properties available is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of elements in the `pProperties` array, and on return the variable is overwritten with the number of structures actually written to `pProperties`. If `pPropertyCount` is less than the number of layer properties available, at most `pPropertyCount` structures will be written, and `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available properties were returned.

The list of layers enumerated by `vkEnumerateDeviceLayerProperties` must be exactly the sequence of layers enabled for the instance. The members of `VkLayerProperties` for each enumerated layer must be the same as the properties when the layer was enumerated by `vkEnumerateInstanceLayerProperties`.
### Valid Usage (Implicit)

- VUID-vkEnumerateDeviceLayerProperties-physicalDevice-parameter
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkEnumerateDeviceLayerProperties-pPropertyCount-parameter
  - `pPropertyCount` must be a valid pointer to a `uint32_t` value
- VUID-vkEnumerateDeviceLayerProperties-pProperties-parameter
  - If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not `NULL`, `pProperties` must be a valid pointer to an array of `pPropertyCount` `VkLayerProperties` structures

### Return Codes

#### Success

- `VK_SUCCESS`
- `VK_INCOMPLETE`

#### Failure

- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `ppEnabledLayerNames` and `enabledLayerCount` members of `VkDeviceCreateInfo` are deprecated and their values must be ignored by implementations. However, for compatibility, only an empty list of layers or a list that exactly matches the sequence enabled at instance creation time are valid, and validation layers should issue diagnostics for other cases.

Regardless of the enabled layer list provided in `VkDeviceCreateInfo`, the sequence of layers active for a device will be exactly the sequence of layers enabled when the parent instance was created.

### 40.4. Extensions

Extensions may define new Vulkan commands, structures, and enumerants. For compilation purposes, the interfaces defined by registered extensions, including new structures and enumerants as well as function pointer types for new commands, are defined in the Khronos-supplied `vulkan_core.h` together with the core API. However, commands defined by extensions may not be available for static linking - in which case function pointers to these commands should be queried at runtime as described in Command Function Pointers. Extensions may be provided by layers as well as by a Vulkan implementation.

Because extensions may extend or change the behavior of the Vulkan API, extension authors should add support for their extensions to the Khronos validation layers. This is especially important for new commands whose parameters have been wrapped by the validation layers. See the “Vulkan Loader Specification and Architecture Overview” document for additional information.
Note

To enable an instance extension, the name of the extension can be added to the `ppEnabledExtensionNames` member of `VkInstanceCreateInfo` when creating a `VkInstance`.

To enable a device extension, the name of the extension can be added to the `ppEnabledExtensionNames` member of `VkDeviceCreateInfo` when creating a `VkDevice`.

Physical-Device-Level functionality does not have any enabling mechanism and can be used as long as the `VkPhysicalDevice` supports the device extension as determined by `vkEnumerateDeviceExtensionProperties`.

Enabling an extension (with no further use of that extension) does not change the behavior of functionality exposed by the core Vulkan API or any other extension, other than making valid the use of the commands, enums and structures defined by that extension.

Valid Usage sections for individual commands and structures do not currently contain which extensions have to be enabled in order to make their use valid, although they might do so in the future. It is defined only in the Valid Usage for Extensions section.

### 40.4.1. Instance Extensions

Instance extensions add new instance-level functionality to the API, outside of the core specification.

To query the available instance extensions, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkEnumerateInstanceExtensionProperties(
    const char* pLayerName,
    uint32_t* pPropertyCount,
    VkExtensionProperties* pProperties);
```

- `pLayerName` is either NULL or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, as described below.
- `pProperties` is either NULL or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is NULL, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the instance extensions provided by that layer are returned.

If `pProperties` is NULL, then the number of extensions properties available is returned in `pPropertyCount`. Otherwise, `pPropertyCount` must point to a variable set by the user to the number of
elements in the \textit{pProperties} array, and on return the variable is overwritten with the number of structures actually written to \textit{pProperties}. If \textit{pPropertyCount} is less than the number of extension properties available, at most \textit{pPropertyCount} structures will be written, and \textit{VK_INCOMPLETE} will be returned instead of \textit{VK_SUCCESS}, to indicate that not all the available properties were returned.

Because the list of available layers may change externally between calls to \textit{vkEnumerateInstanceExtensionProperties}, two calls may retrieve different results if a \textit{pLayerName} is available in one call but not in another. The extensions supported by a layer may also change between two calls, e.g. if the layer implementation is replaced by a different version between those calls.

Implementations \textbf{must} not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

### Valid Usage (Implicit)

- VUID-vkEnumerateInstanceExtensionProperties-pLayerName-parameter
  
  If \textit{pLayerName} is not NULL, \textit{pLayerName} \textbf{must} be a null-terminated UTF-8 string

- VUID-vkEnumerateInstanceExtensionProperties-pPropertyCount-parameter
  
  \textit{pPropertyCount} \textbf{must} be a valid pointer to a uint32_t value

- VUID-vkEnumerateInstanceExtensionProperties-pProperties-parameter
  
  If the value referenced by \textit{pPropertyCount} is not 0, and \textit{pProperties} is not NULL, \textit{pProperties} \textbf{must} be a valid pointer to an array of \textit{pPropertyCount} VkExtensionProperties structures

### Return Codes

#### Success

- VK_SUCCESS
- VK_INCOMPLETE

#### Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_LAYER_NOT_PRESENT

### 40.4.2. Device Extensions

Device extensions add new \textit{device-level functionality} to the API, outside of the core specification.

To query the extensions available to a given physical device, call:
// Provided by VK_VERSION_1_0
VkResult vkEnumerateDeviceExtensionProperties(
    VkPhysicalDevice                            physicalDevice,
    const char*                                 pLayerName,
    uint32_t*                                   pPropertyCount,
    VkExtensionProperties*                      pProperties);

- `physicalDevice` is the physical device that will be queried.
- `pLayerName` is either `NULL` or a pointer to a null-terminated UTF-8 string naming the layer to retrieve extensions from.
- `pPropertyCount` is a pointer to an integer related to the number of extension properties available or queried, and is treated in the same fashion as the `vkEnumerateInstanceExtensionProperties::pPropertyCount` parameter.
- `pProperties` is either `NULL` or a pointer to an array of `VkExtensionProperties` structures.

When `pLayerName` parameter is `NULL`, only extensions provided by the Vulkan implementation or by implicitly enabled layers are returned. When `pLayerName` is the name of a layer, the device extensions provided by that layer are returned.

Implementations **must** not advertise any pair of extensions that cannot be enabled together due to behavioral differences, or any extension that cannot be enabled against the advertised version.

### Valid Usage (Implicit)

- **VUID-vkEnumerateDeviceExtensionProperties-physicalDevice-parameter**  
  `physicalDevice` **must** be a valid `VkPhysicalDevice` handle

- **VUID-vkEnumerateDeviceExtensionProperties-pLayerName-parameter**  
  If `pLayerName` is not `NULL`, `pLayerName` **must** be a null-terminated UTF-8 string

- **VUID-vkEnumerateDeviceExtensionProperties-pPropertyCount-parameter**  
  `pPropertyCount` **must** be a valid pointer to a `uint32_t` value

- **VUID-vkEnumerateDeviceExtensionProperties-pProperties-parameter**  
  If the value referenced by `pPropertyCount` is not 0, and `pProperties` is not `NULL`, `pProperties` **must** be a valid pointer to an array of `pPropertyCount` `VkExtensionProperties` structures
Return Codes

**Success**
- VK_SUCCESS
- VK_INCOMPLETE

**Failure**
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_LAYER_NOT_PRESENT

The `VkExtensionProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkExtensionProperties {
    char    extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t specVersion;
} VkExtensionProperties;
```

- `extensionName` is an array of `VK_MAX_EXTENSION_NAME_SIZE` `char` containing a null-terminated UTF-8 string which is the name of the extension.
- `specVersion` is the version of this extension. It is an integer, incremented with backward compatible changes.

### 40.5. Extension Dependencies

Some extensions are dependent on other extensions, or on specific core API versions, to function. To enable extensions with dependencies, any *required extensions* must also be enabled through the same API mechanisms when creating an instance with `vkCreateInstance` or a device with `vkCreateDevice`. Each extension which has such dependencies documents them in the appendix summarizing that extension.

If an extension is supported (as queried by `vkEnumerateInstanceExtensionProperties` or `vkEnumerateDeviceExtensionProperties`), then *required extensions* of that extension must also be supported for the same instance or physical device.

Any device extension that has an instance extension dependency that is not enabled by `vkCreateInstance` is considered to be unsupported, hence it must not be returned by `vkEnumerateDeviceExtensionProperties` for any `VkPhysicalDevice` child of the instance.

If a required extension has been *promoted* to another extension or to a core API version, then as a *general* rule, the dependency is also satisfied by the promoted extension or core version. This will be true so long as any features required by the original extension are also required or enabled by the promoted extension or core version. However, in some cases an extension is promoted while making some of its features optional in the promoted extension or core version. In this case, the
dependency may not be satisfied. The only way to be certain is to look at the descriptions of the original dependency and the promoted version in the Layers & Extensions and Core Revisions appendices.

Note
There is metadata in \texttt{vk.xml} describing some aspects of promotion, especially requires, promotedto and deprecatedby attributes of \texttt{<extension>} tags. However, the metadata does not yet fully describe this scenario. In the future, we may extend the XML schema to describe the full set of extensions and versions satisfying a dependency.

40.6. Compatibility Guarantees (Informative)

This section is marked as informal as there is no binding responsibility on implementations of the Vulkan API - these guarantees are however a contract between the Vulkan Working Group and developers using this Specification.

40.6.1. Core Versions

Each of the major, minor, and patch versions of the Vulkan specification provide different compatibility guarantees.

Patch Versions

A difference in the patch version indicates that a set of bug fixes or clarifications have been made to the Specification. Informative enums returned by Vulkan commands that will not affect the runtime behavior of a valid application may be added in a patch version (e.g. \texttt{VkVendorId}).

The specification's patch version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the patch version being increased by 1. Patch versions are applied to all minor versions, even if a given minor version is not affected by the provoking change.

Specifications with different patch versions but the same major and minor version are \textit{fully compatible} with each other - such that a valid application written against one will work with an implementation of another.

Note

If a patch version includes a bug fix or clarification that could have a significant impact on developer expectations, these will be highlighted in the change log. Generally the Vulkan Working Group tries to avoid these kinds of changes, instead fixing them in either an extension or core version.

Minor Versions

Changes in the minor version of the specification indicate that new functionality has been added to the core specification. This will usually include new interfaces in the header, and may also include behavior changes and bug fixes. Core functionality may be deprecated in a minor version, but will
not be obsoleted or removed.

The specification’s minor version is strictly increasing for a given major version of the specification; any change to a specification as described above will result in the minor version being increased by 1. Changes that can be accommodated in a patch version will not increase the minor version.

Specifications with a lower minor version are *backwards compatible* with an implementation of a specification with a higher minor version for core functionality and extensions issued with the KHR vendor tag. Vendor and multi-vendor extensions are not guaranteed to remain functional across minor versions, though in general they are with few exceptions - see Obsoletion for more information.

**Major Versions**

A difference in the major version of specifications indicates a large set of changes which will likely include interface changes, behavioral changes, removal of deprecated functionality, and the modification, addition, or replacement of other functionality.

The specification’s major version is monotonically increasing; any change to the specification as described above will result in the major version being increased. Changes that can be accommodated in a patch or minor version will not increase the major version.

The Vulkan Working Group intends to only issue a new major version of the Specification in order to realise significant improvements to the Vulkan API that will necessarily require breaking compatibility.

A new major version will likely include a wholly new version of the specification to be issued - which could include an overhaul of the versioning semantics for the minor and patch versions. The patch and minor versions of a specification are therefore not meaningful across major versions. If a major version of the specification includes similar versioning semantics, it is expected that the patch and the minor version will be reset to 0 for that major version.

**40.6.2. Extensions**

A KHR extension must be able to be enabled alongside any other KHR extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A multi-vendor extension should be able to be enabled alongside any KHR extension or other multi-vendor extension, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension should be able to be enabled alongside any KHR extension, multi-vendor extension, or other vendor extension from the same vendor, and for any minor or patch version of the core Specification beyond the minimum version it requires. A vendor extension may be able to be enabled alongside vendor extensions from another vendor.

The one other exception to this is if a vendor or multi-vendor extension is made obsolete by either a core version or another extension, which will be highlighted in the extension appendix.

**Promotion**

Extensions, or features of an extension, may be promoted to a new core version of the API, or a
newer extension which an equal or greater number of implementors are in favour of.

When extension functionality is promoted, minor changes may be introduced, limited to the following:

- Naming
- Non-intrusive parameters changes
- Feature advertisement/enablement
- Combining structure parameters into larger structures
- Author ID suffixes changed or removed

**Note**

If extension functionality is promoted, there is no guarantee of direct compatibility, however it should require little effort to port code from the original feature to the promoted one.

The Vulkan Working Group endeavours to ensure that larger changes are marked as either deprecated or obsoleted as appropriate, and can do so retroactively if necessary.

Extensions that are promoted are listed as being promoted in their extension appendices, with reference to where they were promoted to.

When an extension is promoted, any backwards compatibility aliases which exist in the extension will not be promoted.

**Note**

As a hypothetical example, if the VK_KHR_surface extension were promoted to part of a future core version, the VK_COLOR_SPACE_SRGB_NONLINEAR_KHR token defined by that extension would be promoted to VK_COLOR_SPACE_SRGB_NONLINEAR. However, the VK_COLORSPACE_SRGB_NONLINEAR_KHR token aliases VK_COLOR_SPACE_SRGB_NONLINEAR_KHR. The VK_COLORSPACE_SRGB_NONLINEAR_KHR token would not be promoted, because it is a backwards compatibility alias that exists only due to a naming mistake when the extension was initially published.

**Deprecation**

Extensions may be marked as deprecated when the intended use cases either become irrelevant or can be solved in other ways. Generally, a new feature will become available to solve the use case in another extension or core version of the API, but it is not guaranteed.

**Note**

Features that are intended to replace deprecated functionality have no guarantees of compatibility, and applications may require drastic modification in order to make use of the new features.
Extensions that are deprecated are listed as being deprecated in their extension appendices, with an explanation of the deprecation and any features that are relevant.

**Obsoletion**

Occasionally, an extension will be marked as obsolete if a new version of the core API or a new extension is fundamentally incompatible with it. An obsoleted extension must not be used with the extension or core version that obsoleted it.

Extensions that are obsoleted are listed as being obsoleted in their extension appendices, with reference to what they were obsoleted by.

**Aliases**

When an extension is promoted or deprecated by a newer feature, some or all of its functionality may be replicated into the newer feature. Rather than duplication of all the documentation and definitions, the specification instead identifies the identical commands and types as aliases of one another. Each alias is mentioned together with the definition it aliases, with the older aliases marked as “equivalents”. Each alias of the same command has identical behavior, and each alias of the same type has identical meaning - they can be used interchangeably in an application with no compatibility issues.

*Note*

For promoted types, the aliased extension type is semantically identical to the new core type. The C99 headers simply typedef the older aliases to the promoted types.

For promoted command aliases, however, there are two separate entry point definitions, due to the fact that the C99 ABI has no way to alias command definitions without resorting to macros. Calling via either entry point definition will produce identical behavior within the bounds of the specification, and should still invoke the same entry point in the implementation. Debug tools may use separate entry points with different debug behavior; to write the appropriate command name to an output log, for instance.

**Special Use Extensions**

Some extensions exist only to support a specific purpose or specific class of application. These are referred to as “special use extensions”. Use of these extensions in applications not meeting the special use criteria is not recommended.

Special use cases are restricted, and only those defined below are used to describe extensions:

*Table 51. Extension Special Use Cases*

<table>
<thead>
<tr>
<th>Special Use</th>
<th>XML Tag</th>
<th>Full Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD support</td>
<td>cadsupport</td>
<td>Extension is intended to support specialized functionality used by CAD/CAM applications.</td>
</tr>
<tr>
<td>Special Use</td>
<td>XML Tag</td>
<td>Full Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>D3D support</td>
<td>d3demulation</td>
<td>Extension is intended to support D3D emulation layers, and applications ported from D3D, by adding functionality specific to D3D.</td>
</tr>
<tr>
<td>Developer tools</td>
<td>devtools</td>
<td>Extension is intended to support developer tools such as capture-replay libraries.</td>
</tr>
<tr>
<td>Debugging tools</td>
<td>debugging</td>
<td>Extension is intended for use by applications when debugging.</td>
</tr>
<tr>
<td>OpenGL / ES support</td>
<td>glemulation</td>
<td>Extension is intended to support OpenGL and/or OpenGL ES emulation layers, and applications ported from those APIs, by adding functionality specific to those APIs.</td>
</tr>
</tbody>
</table>

Special use extensions are identified in the metadata for each such extension in the Layers & Extensions appendix, using the name in the “Special Use” column above.

Special use extensions are also identified in vk.xml with the short name in “XML Tag” column above, as described in the “API Extensions (extension tag)” section of the registry schema documentation.

If the VkTraceRaysIndirectCommandKHR structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkTraceRaysIndirectCommandKHR can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.
Chapter 41. Features

Features describe functionality which is not supported on all implementations. Features are properties of the physical device. Features are **optional**, and **must** be explicitly enabled before use. Support for features is reported and enabled on a per-feature basis.

### Note

Features are reported via the basic `VkPhysicalDeviceFeatures` structure, as well as the extensible structure `VkPhysicalDeviceFeatures2`, which was added in the `VK_KHR_get_physical_device_properties2` extension and included in Vulkan 1.1. When new features are added in future Vulkan versions or extensions, each extension **should** introduce one new feature structure, if needed. This structure can be added to the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure.

For convenience, new core versions of Vulkan **may** introduce new unified feature structures for features promoted from extensions. At the same time, the extension's original feature structure (if any) is also promoted to the core API, and is an alias of the extension's structure. This results in multiple names for the same feature: in the original extension's feature structure and the promoted structure alias, in the unified feature structure. When a feature was implicitly supported and enabled in the extension, but an explicit name was added during promotion, then the extension itself acts as an alias for the feature as listed in the table below.

All aliases of the same feature in the core API **must** be reported consistently: either all **must** be reported as supported, or none of them. When a promoted extension is available, any corresponding feature aliases **must** be supported.

<table>
<thead>
<tr>
<th>Table 52. Extension Feature Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension</td>
</tr>
</tbody>
</table>

To query supported features, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceFeatures(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceFeatures* pFeatures);
```

- **physicalDevice** is the physical device from which to query the supported features.
- **pFeatures** is a pointer to a `VkPhysicalDeviceFeatures` structure in which the physical device features are returned. For each feature, a value of `VK_TRUE` specifies that the feature is supported on this physical device, and `VK_FALSE` specifies that the feature is not supported.
Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceFeatures-physicalDevice-parameter
  physicalDevice must be a valid VkPhysicalDevice handle

- VUID-vkGetPhysicalDeviceFeatures-pFeatures-parameter
  pFeatures must be a valid pointer to a VkPhysicalDeviceFeatures structure

Fine-grained features used by a logical device must be enabled at VkDevice creation time. If a feature is enabled that the physical device does not support, VkDevice creation will fail and return VK_ERROR_FEATURE_NOT_PRESENT.

The fine-grained features are enabled by passing a pointer to the VkPhysicalDeviceFeatures structure via the pEnabledFeatures member of the VkDeviceCreateInfo structure that is passed into the vkCreateDevice call. If a member of pEnabledFeatures is set to VK_TRUE or VK_FALSE, then the device will be created with the indicated feature enabled or disabled, respectively. Features can also be enabled by using the VkPhysicalDeviceFeatures2 structure.

If an application wishes to enable all features supported by a device, it can simply pass in the VkPhysicalDeviceFeatures structure that was previously returned by vkGetPhysicalDeviceFeatures. To disable an individual feature, the application can set the desired member to VK_FALSE in the same structure. Setting pEnabledFeatures to NULL and not including a VkPhysicalDeviceFeatures2 in the pNext chain of VkDeviceCreateInfo is equivalent to setting all members of the structure to VK_FALSE.

Note
Some features, such as robustBufferAccess, may incur a runtime performance cost. Application writers should carefully consider the implications of enabling all supported features.

To query supported features defined by the core or extensions, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceFeatures2(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceFeatures2* pFeatures);
```

or the equivalent command

```c
// Provided by VK_KHR_get_physical_device_properties2
void vkGetPhysicalDeviceFeatures2KHR(
    VkPhysicalDevice physicalDevice,
    VkPhysicalDeviceFeatures2* pFeatures);
```

- physicalDevice is the physical device from which to query the supported features.
- pFeatures is a pointer to a VkPhysicalDeviceFeatures2 structure in which the physical device
features are returned.

Each structure in `pFeatures` and its `pNext` chain contains members corresponding to fine-grained features. `vkGetPhysicalDeviceFeatures2` writes each member to a boolean value indicating whether that feature is supported.

### Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceFeatures2-physicalDevice-parameter`
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `VUID-vkGetPhysicalDeviceFeatures2-pFeatures-parameter`
  - `pFeatures` must be a valid pointer to a `VkPhysicalDeviceFeatures2` structure

The `VkPhysicalDeviceFeatures2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceFeatures2 {
    VkStructureType             sType;
    void*                       pNext;
    VkPhysicalDeviceFeatures    features;
} VkPhysicalDeviceFeatures2;
```

or the equivalent

```c
// Provided by VK_KHR_get_physical_device_properties2
typedef VkPhysicalDeviceFeatures2 VkPhysicalDeviceFeatures2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `features` is a `VkPhysicalDeviceFeatures` structure describing the fine-grained features of the Vulkan 1.0 API.

The `pNext` chain of this structure is used to extend the structure with features defined by extensions. This structure can be used in `vkGetPhysicalDeviceFeatures2` or can be included in the `pNext` chain of a `VkDeviceCreateInfo` structure, in which case it controls which features are enabled in the device in lieu of `pEnabledFeatures`.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceFeatures2-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2`

The `VkPhysicalDeviceFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
```
typedef struct VkPhysicalDeviceFeatures {
    VkBool32 robustBufferAccess;
    VkBool32 fullDrawIndexUint32;
    VkBool32 imageCubeArray;
    VkBool32 independentBlend;
    VkBool32 geometryShader;
    VkBool32 tessellationShader;
    VkBool32 sampleRateShading;
    VkBool32 dualSrcBlend;
    VkBool32 logicOp;
    VkBool32 multiDrawIndirect;
    VkBool32 drawIndirectFirstInstance;
    VkBool32 depthClamp;
    VkBool32 depthBiasClamp;
    VkBool32 fillModeNonSolid;
    VkBool32 depthBounds;
    VkBool32 wideLines;
    VkBool32 largePoints;
    VkBool32 alphaToOne;
    VkBool32 multiViewport;
    VkBool32 samplerAnisotropy;
    VkBool32 textureCompressionETC2;
    VkBool32 textureCompressionASTC_LDR;
    VkBool32 textureCompressionBC;
    VkBool32 occlusionQueryPrecise;
    VkBool32 pipelineStatisticsQuery;
    VkBool32 vertexPipelineStoresAndAtomics;
    VkBool32 fragmentStoresAndAtomics;
    VkBool32 shaderTessellationAndGeometryPointSize;
    VkBool32 shaderImageGatherExtended;
    VkBool32 shaderStorageImageExtendedFormats;
    VkBool32 shaderStorageImageMultisample;
    VkBool32 shaderStorageImageReadWithoutFormat;
    VkBool32 shaderStorageImageWriteWithoutFormat;
    VkBool32 shaderUniformBufferArrayDynamicIndexing;
    VkBool32 shaderSampledImageArrayDynamicIndexing;
    VkBool32 shaderStorageBufferArrayDynamicIndexing;
    VkBool32 shaderStorageImageArrayDynamicIndexing;
    VkBool32 shaderClipDistance;
    VkBool32 shaderCullDistance;
    VkBool32 shaderFloat64;
    VkBool32 shaderInt64;
    VkBool32 shaderInt16;
    VkBool32 shaderResourceResidency;
    VkBool32 shaderResourceMinLod;
    VkBool32 sparseBinding;
    VkBool32 sparseResidencyBuffer;
    VkBool32 sparseResidencyImage2D;
    VkBool32 sparseResidencyImage3D;
    VkBool32 sparseResidency2Samples;
} VkPhysicalDeviceFeatures;
This structure describes the following features:

- **robustBufferAccess** specifies that accesses to buffers are bounds-checked against the range of the buffer descriptor (as determined by VkDescriptorBufferInfo::range, VkBufferViewCreateInfo::range, or the size of the buffer). Out of bounds accesses must not cause application termination, and the effects of shader loads, stores, and atomics must conform to an implementation-dependent behavior as described below.

  - A buffer access is considered to be out of bounds if any of the following are true:

    - The pointer was formed by `OpImageTexelPointer` and the coordinate is less than zero or greater than or equal to the number of whole elements in the bound range.

    - The pointer was not formed by `OpImageTexelPointer` and the object pointed to is not wholly contained within the bound range. This includes accesses performed via variable pointers where the buffer descriptor being accessed cannot be statically determined. Uninitialized pointers and pointers equal to `OpConstantNull` are treated as pointing to a zero-sized object, so all accesses through such pointers are considered to be out of bounds. Buffer accesses through buffer device addresses are not bounds-checked. If the cooperativeMatrixRobustBufferAccess feature is not enabled, then accesses using `OpCooperativeMatrixLoadNV` and `OpCooperativeMatrixStoreNV` may not be bounds-checked.

  - Accesses to descriptors written with a `VK_NULL_HANDLE` resource or view are not considered to be out of bounds. Instead, each type of descriptor access defines a specific behavior for accesses to a null descriptor.

    ```c
    VkBool32 sparseResidency4Samples;
    VkBool32 sparseResidency8Samples;
    VkBool32 sparseResidency16Samples;
    VkBool32 sparseResidencyAliased;
    VkBool32 variableMultisampleRate;
    VkBool32 inheritedQueries;
    } VkPhysicalDeviceFeatures;
    ```
Out-of-bounds buffer loads will return any of the following values:

- If the access is to a uniform buffer and robustBufferAccess2 is enabled, loads of offsets between the end of the descriptor range and the end of the descriptor range rounded up to a multiple of robustUniformBufferAccessSizeAlignment bytes must return either zero values or the contents of the memory at the offset being loaded. Loads of offsets past the descriptor range rounded up to a multiple of robustUniformBufferAccessSizeAlignment bytes must return zero values.

- If the access is to a storage buffer and robustBufferAccess2 is enabled, loads of offsets between the end of the descriptor range and the end of the descriptor range rounded up to a multiple of robustStorageBufferAccessSizeAlignment bytes must return either zero values or the contents of the memory at the offset being loaded. Loads of offsets past the descriptor range rounded up to a multiple of robustStorageBufferAccessSizeAlignment bytes must return zero values. Similarly, stores to addresses between the end of the descriptor range and the end of the descriptor range rounded up to a multiple of robustStorageBufferAccessSizeAlignment bytes may be discarded.

- Non-atomic accesses to storage buffers that are a multiple of 32 bits may be decomposed into 32-bit accesses that are individually bounds-checked.

- If the access is to an index buffer and robustBufferAccess2 is enabled, zero values must be returned.

- If the access is to a uniform texel buffer or storage texel buffer and robustBufferAccess2 is enabled, zero values must be returned, and then Conversion to RGBA is applied based on the buffer view's format.

- Values from anywhere within the memory range(s) bound to the buffer (possibly including bytes of memory past the end of the buffer, up to the end of the bound range).

- Zero values, or (0,0,0,x) vectors for vector reads where x is a valid value represented in the type of the vector components and may be any of:
  - 0, 1, or the maximum representable positive integer value, for signed or unsigned integer components
  - 0.0 or 1.0, for floating-point components

- Out-of-bounds writes may modify values within the memory range(s) bound to the buffer, but must not modify any other memory.
  - If robustBufferAccess2 is enabled, out of bounds writes must not modify any memory.

- Out-of-bounds atomics may modify values within the memory range(s) bound to the buffer, but must not modify any other memory, and return an undefined value.
  - If robustBufferAccess2 is enabled, out of bounds atomics must not modify any memory, and return an undefined value.

- If robustBufferAccess2 is disabled, vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than either:
  - vertexBufferRangeSize, if bindingStride == 0; or
  - (vertexBufferRangeSize - (vertexBufferRangeSize % bindingStride))
where vertexBufferRangeSize is the byte size of the memory range bound to the vertex buffer binding and bindingStride is the byte stride of the corresponding vertex input binding. Further, if any vertex input attribute using a specific vertex input binding is out of bounds, then all vertex input attributes using that vertex input binding for that vertex shader invocation are considered out of bounds.

- If a vertex input attribute is out of bounds, it will be assigned one of the following values:
  - Values from anywhere within the memory range(s) bound to the buffer, converted according to the format of the attribute.
  - Zero values, format converted according to the format of the attribute.
  - Zero values, or (0,0,0,x) vectors, as described above.
- If robustBufferAccess2 is enabled, vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than the byte size of the memory range bound to the vertex buffer binding.
- If robustBufferAccess is not enabled, applications must not perform out of bounds accesses.
- If robustBufferAccess2 is enabled, vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than the byte size of the memory range bound to the vertex buffer binding.
- If a vertex input attribute is out of bounds, the raw data extracted are zero values, and missing G, B, or A components are filled with (0,0,1).
  - If robustBufferAccess2 is enabled, vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than the byte size of the memory range bound to the vertex buffer binding.
- If robustBufferAccess is not enabled, applications must not perform out of bounds accesses.
- If robustBufferAccess2 is enabled, vertex input attributes are considered out of bounds if the offset of the attribute in the bound vertex buffer range plus the size of the attribute is greater than the byte size of the memory range bound to the vertex buffer binding.

- fullDrawIndexUint32 specifies the full 32-bit range of indices is supported for indexed draw calls when using a VkIndexType of VK_INDEX_TYPE_UINT32. maxDrawIndexedIndexValue is the maximum index value that may be used (aside from the primitive restart index, which is always \(2^{32} - 1\) when the VkIndexType is VK_INDEX_TYPE_UINT32). If this feature is supported, maxDrawIndexedIndexValue must be \(2^{32} - 1\); otherwise it must be no smaller than \(2^{24} - 1\). See maxDrawIndexedIndexValue.
- imageCubeArray specifies whether image views with a VkImageViewType of VK_IMAGE_VIEW_TYPE_CUBE_ARRAY can be created, and that the corresponding SampledCubeArray and ImageCubeArray SPIR-V capabilities can be used in shader code.
- independentBlend specifies whether the VkPipelineColorBlendAttachmentState settings are controlled independently per-attachment. If this feature is not enabled, the VkPipelineColorBlendAttachmentState settings for all color attachments must be identical. Otherwise, a different VkPipelineColorBlendAttachmentState can be provided for each bound color attachment.
- geometryShader specifies whether geometry shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_GEOMETRY_BIT and VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT enum values must not be used. This also specifies whether shader modules can declare the Geometry capability.
- tessellationShader specifies whether tessellation control and evaluation shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT, VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, and VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO enum values must not be used. This also specifies whether shader modules can declare the Tessellation capability.
- sampleRateShading specifies whether Sample Shading and multisample interpolation are
supported. If this feature is not enabled, the `sampleShadingEnable` member of the `VkPipelineMultisampleStateCreateInfo` structure **must** be set to `VK_FALSE` and the `minSampleShading` member is ignored. This also specifies whether shader modules **can** declare the `SampleRateShading` capability.

- **dualSrcBlend** specifies whether blend operations which take two sources are supported. If this feature is not enabled, the `VK_BLEND_FACTOR_SRC1_COLOR`, `VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_FACTOR_SRC1_ALPHA`, and `VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA` enum values **must** not be used as source or destination blending factors. See Dual-Source Blending.

- **logicOp** specifies whether logic operations are supported. If this feature is not enabled, the `logicOpEnable` member of the `VkPipelineColorBlendStateCreateInfo` structure **must** be set to `VK_FALSE`, and the `logicOp` member is ignored.

- **multiDrawIndirect** specifies whether multiple draw indirect is supported. If this feature is not enabled, the `drawCount` parameter to the `vkCmdDrawIndirect` and `vkCmdDrawIndexedIndirect` commands **must** be 0 or 1. The `maxDrawIndirectCount` member of the `VkPhysicalDeviceIndirectCountLimits` structure **must** also be 1 if this feature is not supported. See `maxDrawIndirectCount`.

- **drawIndirectFirstInstance** specifies whether indirect draw calls support the `firstInstance` parameter. If this feature is not enabled, the `firstInstance` member of all `VkDrawIndirectCommand` and `VkDrawIndexedIndirectCommand` structures that are provided to the `vkCmdDrawIndirect` and `vkCmdDrawIndexedIndirect` commands **must** be 0.

- **depthClamp** specifies whether depth clamping is supported. If this feature is not enabled, the `depthClampEnable` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to `VK_FALSE`. Otherwise, setting `depthClampEnable` to `VK_TRUE` will enable depth clamping.

- **depthBiasClamp** specifies whether depth bias clamping is supported. If this feature is not enabled, the `depthBiasClamp` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to 0.0 unless the `VK_DYNAMIC_STATE_DEPTH_BIAS` dynamic state is enabled, and the `depthBiasClamp` parameter to `vkCmdSetDepthBias` **must** be set to 0.0.

- **fillModeNonSolid** specifies whether point and wireframe fill modes are supported. If this feature is not enabled, the `VK_POLYGON_MODE_POINT` and `VK_POLYGON_MODE_LINE` enum values **must** not be used.

- **depthBounds** specifies whether depth bounds tests are supported. If this feature is not enabled, the `depthBoundsTestEnable` member of the `VkPipelineDepthStencilStateCreateInfo` structure **must** be set to `VK_FALSE`. When `depthBoundsTestEnable` is set to `VK_FALSE`, the `minDepthBounds` and `maxDepthBounds` members of the `VkPipelineDepthStencilStateCreateInfo` structure are ignored.

- **wideLines** specifies whether lines with width other than 1.0 are supported. If this feature is not enabled, the `lineWidth` member of the `VkPipelineRasterizationStateCreateInfo` structure **must** be set to 1.0 unless the `VK_DYNAMIC_STATE_LINE_WIDTH` dynamic state is enabled, and the `lineWidth` parameter to `vkCmdSetLineWidth` **must** be set to 1.0. When this feature is supported, the range and granularity of supported line widths are indicated by the `lineWidthRange` and `lineWidthGranularity` members of the `VkPhysicalDeviceLimits` structure, respectively.

- **largePoints** specifies whether points with size greater than 1.0 are supported. If this feature is not enabled, only a point size of 1.0 written by a shader is supported. The range and granularity of supported point sizes are indicated by the `pointSizeRange` and `pointSizeGranularity` members of the `VkPhysicalDeviceLimits` structure, respectively.
• **alphaToOne** specifies whether the implementation is able to replace the alpha value of the fragment shader color output in the `Multisample Coverage` fragment operation. If this feature is not enabled, then the `alphaToOneEnable` member of the `VkPipelineMultisampleStateCreateInfo` structure **must** be set to `VK_FALSE`. Otherwise setting `alphaToOneEnable` to `VK_TRUE` will enable alpha-to-one behavior.

• **multiViewport** specifies whether more than one viewport is supported. If this feature is not enabled:
  
  ◦ The `viewportCount` and `scissorCount` members of the `VkPipelineViewportStateCreateInfo` structure **must** be set to 1.
  
  ◦ The `firstViewport` and `viewportCount` parameters to the `vkCmdSetViewport` command **must** be set to 0 and 1, respectively.
  
  ◦ The `firstScissor` and `scissorCount` parameters to the `vkCmdSetScissor` command **must** be set to 0 and 1, respectively.
  
  ◦ The `exclusiveScissorCount` member of the `VkPipelineViewportExclusiveScissorStateCreateInfoNV` structure **must** be set to 0 or 1.
  
  ◦ The `firstExclusiveScissor` and `exclusiveScissorCount` parameters to the `vkCmdSetExclusiveScissorNV` command **must** be set to 0 and 1, respectively.

• **samplerAnisotropy** specifies whether anisotropic filtering is supported. If this feature is not enabled, the `anisotropyEnable` member of the `VkSamplerCreateInfo` structure **must** be `VK_FALSE`.

• **textureCompressionETC2** specifies whether all of the ETC2 and EAC compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features **must** be supported in `optimalTilingFeatures` for the following formats:

  ◦ `VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK`
  
  ◦ `VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK`
  
  ◦ `VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK`
  
  ◦ `VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK`
  
  ◦ `VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK`
  
  ◦ `VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK`
  
  ◦ `VK_FORMAT_EAC_R11_UNORM_BLOCK`
  
  ◦ `VK_FORMAT_EAC_R11_SNORM_BLOCK`
  
  ◦ `VK_FORMAT_EAC_R11G11_UNORM_BLOCK`
  
  ◦ `VK_FORMAT_EAC_R11G11_SNORM_BLOCK`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` **can** be used to check for supported properties of individual formats as normal.

• **textureCompressionASTC_LDR** specifies whether all of the ASTC LDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
features must be supported in optimalTilingFeatures for the following formats:

- VK_FORMAT_ASTC_4x4_UNORM_BLOCK
- VK_FORMAT_ASTC_4x4_SRGB_BLOCK
- VK_FORMAT_ASTC_5x4_UNORM_BLOCK
- VK_FORMAT_ASTC_5x4_SRGB_BLOCK
- VK_FORMAT_ASTC_5x5_UNORM_BLOCK
- VK_FORMAT_ASTC_5x5_SRGB_BLOCK
- VK_FORMAT_ASTC_6x5_UNORM_BLOCK
- VK_FORMAT_ASTC_6x5_SRGB_BLOCK
- VK_FORMAT_ASTC_6x6_UNORM_BLOCK
- VK_FORMAT_ASTC_6x6_SRGB_BLOCK
- VK_FORMAT_ASTC_8x5_UNORM_BLOCK
- VK_FORMAT_ASTC_8x5_SRGB_BLOCK
- VK_FORMAT_ASTC_8x6_UNORM_BLOCK
- VK_FORMAT_ASTC_8x6_SRGB_BLOCK
- VK_FORMAT_ASTC_8x8_UNORM_BLOCK
- VK_FORMAT_ASTC_8x8_SRGB_BLOCK
- VK_FORMAT_ASTC_10x5_UNORM_BLOCK
- VK_FORMAT_ASTC_10x5_SRGB_BLOCK
- VK_FORMAT_ASTC_10x6_UNORM_BLOCK
- VK_FORMAT_ASTC_10x6_SRGB_BLOCK
- VK_FORMAT_ASTC_10x8_UNORM_BLOCK
- VK_FORMAT_ASTC_10x8_SRGB_BLOCK
- VK_FORMAT_ASTC_10x10_UNORM_BLOCK
- VK_FORMAT_ASTC_10x10_SRGB_BLOCK
- VK_FORMAT_ASTC_12x10_UNORM_BLOCK
- VK_FORMAT_ASTC_12x10_SRGB_BLOCK
- VK_FORMAT_ASTC_12x12_UNORM_BLOCK
- VK_FORMAT_ASTC_12x12_SRGB_BLOCK

To query for additional properties, or if the feature is not enabled, vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties can be used to check for supported properties of individual formats as normal.

- textureCompressionBC specifies whether all of the BC compressed texture formats are supported. If this feature is enabled, then the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT, VK_FORMAT_FEATURE_BLIT_SRC_BIT and VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT
features **must** be supported in `optimalTilingFeatures` for the following formats:

- `VK_FORMAT_BC1_RGB_UNORM_BLOCK`
- `VK_FORMAT_BC1_RGB_SRGB_BLOCK`
- `VK_FORMAT_BC1_RGBA_UNORM_BLOCK`
- `VK_FORMAT_BC1_RGBA_SRGB_BLOCK`
- `VK_FORMAT_BC2_UNORM_BLOCK`
- `VK_FORMAT_BC2_SRGB_BLOCK`
- `VK_FORMAT_BC3_UNORM_BLOCK`
- `VK_FORMAT_BC3_SRGB_BLOCK`
- `VK_FORMAT_BC4_UNORM_BLOCK`
- `VK_FORMAT_BC4_SNORM_BLOCK`
- `VK_FORMAT_BC5_UNORM_BLOCK`
- `VK_FORMAT_BC5_SNORM_BLOCK`
- `VK_FORMAT_BC6H_UFLOAT_BLOCK`
- `VK_FORMAT_BC6H_SFLOAT_BLOCK`
- `VK_FORMAT_BC7_UNORM_BLOCK`
- `VK_FORMAT_BC7_SRGB_BLOCK`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

- **occlusionQueryPrecise** specifies whether occlusion queries returning actual sample counts are supported. Occlusion queries are created in a `VkQueryPool` by specifying the `queryType` of `VK_QUERY_TYPE_OCCLUSION` in the `VkQueryPoolCreateInfo` structure which is passed to `vkCreateQueryPool`. If this feature is enabled, queries of this type **can** enable `VK_QUERY_CONTROL_PRECISE_BIT` in the `flags` parameter to `vkCmdBeginQuery`. If this feature is not supported, the implementation supports only boolean occlusion queries. When any samples are passed, boolean queries will return a non-zero result value, otherwise a result value of zero is returned. When this feature is enabled and `VK_QUERY_CONTROL_PRECISE_BIT` is set, occlusion queries will report the actual number of samples passed.

- **pipelineStatisticsQuery** specifies whether the pipeline statistics queries are supported. If this feature is not enabled, queries of type `VK_QUERY_TYPE_PIPELINE_STATISTICS` **cannot** be created, and none of the `VkQueryPipelineStatisticFlagBits` bits **can** be set in the `pipelineStatistics` member of the `VkQueryPoolCreateInfo` structure.

- **vertexPipelineStoresAndAtomics** specifies whether storage buffers and images support stores and atomic operations in the vertex, tessellation, and geometry shader stages. If this feature is not enabled, all storage image, storage texel buffer, and storage buffer variables used by these stages in shader modules **must** be decorated with the `NonWritable` decoration (or the `readonly` memory qualifier in GLSL).

- **fragmentStoresAndAtomics** specifies whether storage buffers and images support stores and
atomic operations in the fragment shader stage. If this feature is not enabled, all storage image, storage texel buffer, and storage buffer variables used by the fragment stage in shader modules must be decorated with the NonWritable decoration (or the readonly memory qualifier in GLSL).

- shaderTessellationAndGeometryPointSize specifies whether the PointSize built-in decoration is available in the tessellation control, tessellation evaluation, and geometry shader stages. If this feature is not enabled, members decorated with the PointSize built-in decoration must not be read from or written to and all points written from a tessellation or geometry shader will have a size of 1.0. This also specifies whether shader modules can declare the TessellationPointSize capability for tessellation control and evaluation shaders, or if the shader modules can declare the GeometryPointSize capability for geometry shaders. An implementation supporting this feature must also support one or both of the tessellationShader or geometryShader features.

- shaderImageGatherExtended specifies whether the extended set of image gather instructions are available in shader code. If this feature is not enabled, the OpImage*Gather instructions do not support the Offset and ConstOffsets operands. This also specifies whether shader modules can declare the ImageGatherExtended capability.

- shaderStorageImageExtendedFormats specifies whether all the “storage image extended formats” below are supported; if this feature is supported, then the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT must be supported in optimalTilingFeatures for the following formats:

  - VK_FORMAT_R16G16_SFLOAT
  - VK_FORMAT_B10G11R11_UFLOAT_PACK32
  - VK_FORMAT_R16_SFLOAT
  - VK_FORMAT_R16G16B16A16_UNORM
  - VK_FORMAT_A2B10G10R10_UNORM_PACK32
  - VK_FORMAT_R16G16B16A16_SNORM
  - VK_FORMAT_R16G16_SNORM
  - VK_FORMAT_R8G8_SNORM
  - VK_FORMAT_R16_SNORM
  - VK_FORMAT_R8_SNORM
  - VK_FORMAT_A2B10G10R10_UINT_PACK32
  - VK_FORMAT_R16G16_UINT
  - VK_FORMAT_R16G16B16A16_SINT
  - VK_FORMAT_R16G16_SINT
  - VK_FORMAT_R8G8_SINT
  - VK_FORMAT_R16_SINT
  - VK_FORMAT_R8_SINT
Note

shaderStorageImageExtendedFormats feature only adds a guarantee of format support, which is specified for the whole physical device. Therefore enabling or disabling the feature via vkCreateDevice has no practical effect.

To query for additional properties, or if the feature is not supported, vkGetPhysicalDeviceFormatProperties and vkGetPhysicalDeviceImageFormatProperties can be used to check for supported properties of individual formats, as usual rules allow.

VK_FORMAT_R32G32_UINT, VK_FORMAT_R32G32_SINT, and VK_FORMAT_R32G32_SFLOAT from StorageImageExtendedFormats SPIR-V capability, are already covered by core Vulkan mandatory format support.

- shaderStorageImageMultisample specifies whether multisampled storage images are supported. If this feature is not enabled, images that are created with a usage that includes VK_IMAGE_USAGE_STORAGE_BIT must be created with samples equal to VK_SAMPLE_COUNT_1_BIT. This also specifies whether shader modules can declare the StorageImageMultisample and ImageMSArray capabilities.

- shaderStorageImageReadWithoutFormat specifies whether storage images require a format qualifier to be specified when reading from storage images. If this feature is not enabled, the OpImageRead instruction must not have an OpTypeImage of Unknown. This also specifies whether shader modules can declare the StorageImageReadWithoutFormat capability.

- shaderStorageImageWriteWithoutFormat specifies whether storage images require a format qualifier to be specified when writing to storage images. If this feature is not enabled, the OpImageWrite instruction must not have an OpTypeImage of Unknown. This also specifies whether shader modules can declare the StorageImageWriteWithoutFormat capability.

- shaderUniformBufferArrayDynamicIndexing specifies whether arrays of uniform buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the UniformBufferArrayDynamicIndexing capability.

- shaderSampledImageArrayDynamicIndexing specifies whether arrays of samplers or sampled images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, or VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the SampledImageArrayDynamicIndexing capability.

- shaderStorageBufferArrayDynamicIndexing specifies whether arrays of storage buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not
enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the `StorageBufferArrayDynamicIndexing` capability.

- `shaderStorageImageArrayDynamicIndexing` specifies whether arrays of storage images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also specifies whether shader modules can declare the `StorageImageArrayDynamicIndexing` capability.

- `shaderClipDistance` specifies whether clip distances are supported in shader code. If this feature is not enabled, any members decorated with the `ClipDistance` built-in decoration must not be read from or written to in shader modules. This also specifies whether shader modules can declare the `ClipDistance` capability.

- `shaderCullDistance` specifies whether cull distances are supported in shader code. If this feature is not enabled, any members decorated with the `CullDistance` built-in decoration must not be read from or written to in shader modules. This also specifies whether shader modules can declare the `CullDistance` capability.

- `shaderFloat64` specifies whether 64-bit floats (doubles) are supported in shader code. If this feature is not enabled, 64-bit floating-point types must not be used in shader code. This also specifies whether shader modules can declare the `Float64` capability. Declaring and using 64-bit floats is enabled for all storage classes that SPIR-V allows with the `Float64` capability.

- `shaderInt64` specifies whether 64-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 64-bit integer types must not be used in shader code. This also specifies whether shader modules can declare the `Int64` capability. Declaring and using 64-bit integers is enabled for all storage classes that SPIR-V allows with the `Int64` capability.

- `shaderInt16` specifies whether 16-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 16-bit integer types must not be used in shader code. This also specifies whether shader modules can declare the `Int16` capability. However, this only enables a subset of the storage classes that SPIR-V allows for the `Int16` SPIR-V capability: Declaring and using 16-bit integers in the `Private`, `Workgroup` (for non-Block variables), and `Function` storage classes is enabled, while declaring them in the interface storage classes (e.g., `UniformConstant`, `Uniform`, `StorageBuffer`, `Input`, `Output`, and `PushConstant`) is not enabled.

- `shaderResourceResidency` specifies whether image operations that return resource residency information are supported in shader code. If this feature is not enabled, the `OpImageSparse*` instructions must not be used in shader code. This also specifies whether shader modules can declare the `SparseResidency` capability. The feature requires at least one of the `sparseResidency*` features to be supported.

- `shaderResourceMinLod` specifies whether image operations specifying the minimum resource LOD are supported in shader code. If this feature is not enabled, the `MinLod` image operand must not be used in shader code. This also specifies whether shader modules can declare the `MinLod` capability.

- `sparseBinding` specifies whether resource memory can be managed at opaque sparse block level instead of at the object level. If this feature is not enabled, resource memory must be bound only on a per-object basis using the `vkBindBufferMemory` and `vkBindImageMemory` commands. In
this case, buffers and images must not be created with VK_BUFFER_CREATE_SPARSE_BINDING_BIT and VK_IMAGE_CREATE_SPARSE_BINDING_BIT set in the flags member of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively. Otherwise resource memory can be managed as described in Sparse Resource Features.

- sparseResidencyBuffer specifies whether the device can access partially resident buffers. If this feature is not enabled, buffers must not be created with VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkBufferCreateInfo structure.

- sparseResidencyImage2D specifies whether the device can access partially resident 2D images with 1 sample per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_1_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidencyImage3D specifies whether the device can access partially resident 3D images. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_3D must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency2Samples specifies whether the physical device can access partially resident 2D images with 2 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_2_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency4Samples specifies whether the physical device can access partially resident 2D images with 4 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_4_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency8Samples specifies whether the physical device can access partially resident 2D images with 8 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_8_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidency16Samples specifies whether the physical device can access partially resident 2D images with 16 samples per pixel. If this feature is not enabled, images with an imageType of VK_IMAGE_TYPE_2D and samples set to VK_SAMPLE_COUNT_16_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the flags member of the VkImageCreateInfo structure.

- sparseResidencyAliased specifies whether the physical device can correctly access data aliased into multiple locations. If this feature is not enabled, the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT enum values must not be used in flags members of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively.

- variableMultisampleRate specifies whether all pipelines that will be bound to a command buffer during a subpass which uses no attachments must have the same value for VkPipelineMultisampleStateCreateInfo::rasterizationSamples. If set to VK_TRUE, the implementation supports variable multisample rates in a subpass which uses no attachments.
set to `VK_FALSE`, then all pipelines bound in such a subpass **must** have the same multisample rate. This has no effect in situations where a subpass uses any attachments.

- `inheritedQueries` specifies whether a secondary command buffer **may** be executed while a query is active.

The `VkPhysicalDeviceVariablePointersFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceVariablePointersFeatures {
    VkStructureType sType;
    void*             pNext;
    VkBool32          variablePointersStorageBuffer;
    VkBool32          variablePointers;
} VkPhysicalDeviceVariablePointersFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_variable_pointers
typedef VkPhysicalDeviceVariablePointersFeatures
    VkPhysicalDeviceVariablePointersFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `variablePointersStorageBuffer` specifies whether the implementation supports the SPIR-V `VariablePointersStorageBuffer` capability. When this feature is not enabled, shader modules **must** not declare the `SPV_KHR_variable_pointers` extension or the `VariablePointersStorageBuffer` capability.
- `variablePointers` specifies whether the implementation supports the SPIR-V `VariablePointers` capability. When this feature is not enabled, shader modules **must** not declare the `VariablePointers` capability.

If the `VkPhysicalDeviceVariablePointersFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceVariablePointersFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage

- `VUID-VkPhysicalDeviceVariablePointersFeatures-variablePointers-01431`

  If `variablePointers` is enabled then `variablePointersStorageBuffer` **must** also be enabled.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceVariablePointersFeatures-sType-sType**
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES`

The `VkPhysicalDeviceMultiviewFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMultiviewFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           multiview;
    VkBool32           multiviewGeometryShader;
    VkBool32           multiviewTessellationShader;
} VkPhysicalDeviceMultiviewFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_multiview
typedef VkPhysicalDeviceMultiviewFeatures VkPhysicalDeviceMultiviewFeaturesKHR;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **multiview** specifies whether the implementation supports multiview rendering within a render pass. If this feature is not enabled, the view mask of each subpass **must** always be zero.
- **multiviewGeometryShader** specifies whether the implementation supports multiview rendering within a render pass, with geometry shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask **must** not include a geometry shader.
- **multiviewTessellationShader** specifies whether the implementation supports multiview rendering within a render pass, with tessellation shaders. If this feature is not enabled, then a pipeline compiled against a subpass with a non-zero view mask **must** not include any tessellation shaders.

If the `VkPhysicalDeviceMultiviewFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceMultiviewFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage

- VUID-VkPhysicalDeviceMultiviewFeatures-multiviewGeometryShader-00580
  If `multiviewGeometryShader` is enabled then `multiview` must also be enabled

- VUID-VkPhysicalDeviceMultiviewFeatures-multiviewTessellationShader-00581
  If `multiviewTessellationShader` is enabled then `multiview` must also be enabled

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMultiviewFeatures-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES`

The `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_shader_atomic_float
typedef struct VkPhysicalDeviceShaderAtomicFloatFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderBufferFloat32Atomics;
    VkBool32           shaderBufferFloat32AtomicAdd;
    VkBool32           shaderBufferFloat64Atomics;
    VkBool32           shaderBufferFloat64AtomicAdd;
    VkBool32           shaderSharedFloat32Atomics;
    VkBool32           shaderSharedFloat32AtomicAdd;
    VkBool32           shaderSharedFloat64Atomics;
    VkBool32           shaderSharedFloat64AtomicAdd;
    VkBool32           shaderImageFloat32Atomics;
    VkBool32           shaderImageFloat32AtomicAdd;
    VkBool32           sparseImageFloat32Atomics;
    VkBool32           sparseImageFloat32AtomicAdd;
} VkPhysicalDeviceShaderAtomicFloatFeaturesEXT;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderBufferFloat32Atomics` indicates whether shaders can perform 32-bit floating-point load, store and exchange atomic operations on storage buffers.
- `shaderBufferFloat32AtomicAdd` indicates whether shaders can perform 32-bit floating-point add atomic operations on storage buffers.
- `shaderBufferFloat64Atomics` indicates whether shaders can perform 64-bit floating-point load, store and exchange atomic operations on storage buffers.
- `shaderBufferFloat64AtomicAdd` indicates whether shaders can perform 64-bit floating-point add...
atomic operations on storage buffers.

- `shaderSharedFloat32Atomics` indicates whether shaders can perform 32-bit floating-point load, store and exchange atomic operations on shared memory.
- `shaderSharedFloat32AtomicAdd` indicates whether shaders can perform 32-bit floating-point add atomic operations on shared memory.
- `shaderSharedFloat64Atomics` indicates whether shaders can perform 64-bit floating-point load, store and exchange atomic operations on shared memory.
- `shaderSharedFloat64AtomicAdd` indicates whether shaders can perform 64-bit floating-point add atomic operations on shared memory.
- `shaderImageFloat32Atomics` indicates whether shaders can perform 32-bit floating-point load, store and exchange atomic image operations.
- `shaderImageFloat32AtomicAdd` indicates whether shaders can perform 32-bit floating-point add atomic image operations.
- `sparseImageFloat32Atomics` indicates whether 32-bit floating-point load, store and exchange atomic operations can be used on sparse images.
- `sparseImageFloat32AtomicAdd` indicates whether 32-bit floating-point add atomic operations can be used on sparse images.

If the `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderAtomicFloatFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceShaderAtomicFloatFeaturesEXT-sType-sType` 
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT`

The `VkPhysicalDeviceShaderAtomicInt64Features` structure is defined as:

```c
typedef struct VkPhysicalDeviceShaderAtomicInt64Features {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderBufferInt64Atomics;
    VkBool32 shaderSharedInt64Atomics;
} VkPhysicalDeviceShaderAtomicInt64Features;
```

or the equivalent
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **shaderBufferInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on buffers.
- **shaderSharedInt64Atomics** indicates whether shaders can perform 64-bit unsigned and signed integer atomic operations on shared memory.

If the `VkPhysicalDeviceShaderAtomicInt64Features` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderAtomicInt64Features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderAtomicInt64Features-sType-sType
  
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES`.

The `VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT` structure is defined as:

```c
typedef struct VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderImageInt64Atomics;
    VkBool32           sparseImageInt64Atomics;
} VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **shaderImageInt64Atomics** indicates whether shaders can support 64-bit unsigned and signed integer atomic operations on images.
- **sparseImageInt64Atomics** indicates whether 64-bit integer atomics can be used on sparse images.

If the `VkPhysicalDeviceShaderAtomicInt64FeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to
indicate whether each corresponding feature is supported. 

VkPhysicalDeviceShaderAtomicInt64FeaturesEXT can also be used in the pNext chain of 
VkDeviceCreateInfo to selectively enable these features.

<table>
<thead>
<tr>
<th>Valid Usage (Implicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• VUID-VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT-sType-sType</td>
</tr>
<tr>
<td>sType must be</td>
</tr>
<tr>
<td>VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_ATOMIC_INT64_FEATURES_EXT</td>
</tr>
</tbody>
</table>

The VkPhysicalDevice8BitStorageFeatures structure is defined as:

```c
typedef struct VkPhysicalDevice8BitStorageFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           storageBuffer8BitAccess;
    VkBool32           uniformAndStorageBuffer8BitAccess;
    VkBool32           storagePushConstant8;
} VkPhysicalDevice8BitStorageFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_8bit_storage
typedef VkPhysicalDevice8BitStorageFeatures VkPhysicalDevice8BitStorageFeaturesKHR;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **storageBuffer8BitAccess** indicates whether objects in the StorageBuffer, ShaderRecordBufferKHR, or PhysicalStorageBuffer storage class with the Block decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the StorageBuffer8BitAccess capability.
- **uniformAndStorageBuffer8BitAccess** indicates whether objects in the Uniform storage class with the Block decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the UniformAndStorageBuffer8BitAccess capability.
- **storagePushConstant8** indicates whether objects in the PushConstant storage class can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the StoragePushConstant8 capability.

If the VkPhysicalDevice8BitStorageFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to
indicate whether each corresponding feature is supported. `VkPhysicalDevice8BitStorageFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDevice8BitStorageFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES`

The `VkPhysicalDevice16BitStorageFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDevice16BitStorageFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           storageBuffer16BitAccess;
    VkBool32           uniformAndStorageBuffer16BitAccess;
    VkBool32           storagePushConstant16;
    VkBool32           storageInputOutput16;
} VkPhysicalDevice16BitStorageFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_16bit_storage
typedef VkPhysicalDevice16BitStorageFeatures VkPhysicalDevice16BitStorageFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `storageBuffer16BitAccess` specifies whether objects in the `StorageBuffer`, `ShaderRecordBufferKHR`, or `PhysicalStorageBuffer` storage class with the `Block` decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the `StorageBuffer16BitAccess` capability.
- `uniformAndStorageBuffer16BitAccess` specifies whether objects in the `Uniform` storage class with the `Block` decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the `UniformAndStorageBuffer16BitAccess` capability.
- `storagePushConstant16` specifies whether objects in the `PushConstant` storage class can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or floating-point members must not be used in such objects. This also specifies whether shader modules can declare the `StoragePushConstant16` capability.
- `storageInputOutput16` specifies whether objects in the `Input` and `Output` storage classes can have
16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also specifies whether shader modules can declare the StorageInputOutput16 capability.

If the VkPhysicalDevice16BitStorageFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDevice16BitStorageFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDevice16BitStorageFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES

The VkPhysicalDeviceShaderFloat16Int8Features structure is defined as:

```c
typedef struct VkPhysicalDeviceShaderFloat16Int8Features {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderFloat16;
    VkBool32           shaderInt8;
} VkPhysicalDeviceShaderFloat16Int8Features;
```

or the equivalent

```c
// Provided by VK_KHR_shader_float16_int8
typedef VkPhysicalDeviceShaderFloat16Int8Features
VkPhysicalDeviceShaderFloat16Int8FeaturesKHR;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderFloat16** indicates whether 16-bit floats (halfs) are supported in shader code. This also indicates whether shader modules can declare the Float16 capability. However, this only enables a subset of the storage classes that SPIR-V allows for the Float16 SPIR-V capability: Declaring and using 16-bit floats in the Private, Workgroup (for non-Block variables), and Function storage classes is enabled, while declaring them in the interface storage classes (e.g., UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.
- **shaderInt8** indicates whether 8-bit integers (signed and unsigned) are supported in shader code. This also indicates whether shader modules can declare the Int8 capability. However, this only enables a subset of the storage classes that SPIR-V allows for the Int8 SPIR-V capability: Declaring and using 8-bit integers in the Private, Workgroup (for non-Block variables), and Function storage classes is enabled, while declaring them in the interface storage classes (e.g.,
UniformConstant, Uniform, StorageBuffer, Input, Output, and PushConstant) is not enabled.

If the \texttt{VkPhysicalDeviceShaderFloat16Int8Features} structure is included in the \texttt{pNext} chain of the \texttt{VkPhysicalDeviceFeatures2} structure passed to \texttt{vkGetPhysicalDeviceFeatures2}, it is filled in to indicate whether each corresponding feature is supported. \texttt{VkPhysicalDeviceShaderFloat16Int8Features} can also be used in the \texttt{pNext} chain of \texttt{VkDeviceCreateInfo} to selectively enable these features.

**Valid Usage (Implicit)**

- \texttt{VUID-VkPhysicalDeviceShaderFloat16Int8Features-sType-sType} \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES}

The \texttt{VkPhysicalDeviceShaderClockFeaturesKHR} structure is defined as:

```c
// Provided by VK_KHR_shader_clock
typedef struct VkPhysicalDeviceShaderClockFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderSubgroupClock;
    VkBool32 shaderDeviceClock;
} VkPhysicalDeviceShaderClockFeaturesKHR;
```

This structure describes the following features:

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is NULL or a pointer to a structure extending this structure.
- \texttt{shaderSubgroupClock} indicates whether shaders can perform Subgroup scoped clock reads.
- \texttt{shaderDeviceClock} indicates whether shaders can perform Device scoped clock reads.

If the \texttt{VkPhysicalDeviceShaderClockFeaturesKHR} structure is included in the \texttt{pNext} chain of the \texttt{VkPhysicalDeviceFeatures2} structure passed to \texttt{vkGetPhysicalDeviceFeatures2}, it is filled in to indicate whether each corresponding feature is supported. \texttt{VkPhysicalDeviceShaderClockFeaturesKHR} can also be used in the \texttt{pNext} chain of \texttt{VkDeviceCreateInfo} to selectively enable these features.

**Valid Usage (Implicit)**

- \texttt{VUID-VkPhysicalDeviceShaderClockFeaturesKHR-sType-sType} \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR}

The \texttt{VkPhysicalDeviceSamplerYcbcrConversionFeatures} structure is defined as:
This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `samplerYcbcrConversion` specifies whether the implementation supports sampler Y'CbCr conversion. If `samplerYcbcrConversion` is VK_FALSE, sampler Y'CbCr conversion is not supported, and samplers using sampler Y'CbCr conversion must not be used.

If the `VkPhysicalDeviceSamplerYcbcrConversionFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceSamplerYcbcrConversionFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceSamplerYcbcrConversionFeatures-sType-sType`

  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES`.

The `VkPhysicalDeviceProtectedMemoryFeatures` structure is defined as:

```
typedef struct VkPhysicalDeviceProtectedMemoryFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 protectedMemory;
} VkPhysicalDeviceProtectedMemoryFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `protectedMemory` specifies whether protected memory is supported.

If the `VkPhysicalDeviceProtectedMemoryFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceProtectedMemoryFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceProtectedMemoryFeatures-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_FEATURES`

The `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_blend_operation_advanced
typedef struct VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           advancedBlendCoherentOperations;
} VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT;
```

This structure describes the following feature:

• `sType` is the type of this structure.
• `pNext` is NULL or a pointer to a structure extending this structure.
• `advancedBlendCoherentOperations` specifies whether blending using advanced blend operations is guaranteed to execute atomically and in primitive order. If this is `VK_TRUE`, `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is treated the same as `VK_ACCESS_COLOR_ATTACHMENT_READ_BIT`, and advanced blending needs no additional synchronization over basic blending. If this is `VK_FALSE`, then memory dependencies are required to guarantee order between two advanced blending operations that occur on the same sample.

If the `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT`
The `VkPhysicalDeviceConditionalRenderingFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_conditional_rendering
typedef struct VkPhysicalDeviceConditionalRenderingFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           conditionalRendering;
    VkBool32           inheritedConditionalRendering;
} VkPhysicalDeviceConditionalRenderingFeaturesEXT;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `conditionalRendering` specifies whether conditional rendering is supported.
- `inheritedConditionalRendering` specifies whether a secondary command buffer can be executed while conditional rendering is active in the primary command buffer.

If the `VkPhysicalDeviceConditionalRenderingFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceConditionalRenderingFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceConditionalRenderingFeaturesEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONDITIONAL_RENDERING_FEATURES_EXT`

The `VkPhysicalDeviceShaderDrawParametersFeatures` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceShaderDrawParametersFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderDrawParameters;
} VkPhysicalDeviceShaderDrawParametersFeatures;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderDrawParameters` specifies whether shader draw parameters are supported.

If the `VkPhysicalDeviceShaderDrawParametersFeatures` structure is included in the `pNext` chain of the
The VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported.

VkPhysicalDeviceShaderDrawParametersFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderDrawParametersFeatures-sType-sType

  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DRAW_PARAMETERS_FEATURES

The VkPhysicalDeviceMeshShaderFeaturesNV structure is defined as:

```c
// Provided by VK_NV_mesh_shader
typedef struct VkPhysicalDeviceMeshShaderFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           taskShader;
    VkBool32           meshShader;
} VkPhysicalDeviceMeshShaderFeaturesNV;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **taskShader** indicates whether the task shader stage is supported.
- **meshShader** indicates whether the mesh shader stage is supported.

If the VkPhysicalDeviceMeshShaderFeaturesNV structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceMeshShaderFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMeshShaderFeaturesNV-sType-sType

  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_FEATURES_NV

The VkPhysicalDeviceDescriptorIndexingFeatures structure is defined as:
typedef struct VkPhysicalDeviceDescriptorIndexingFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderInputAttachmentArrayDynamicIndexing;
    VkBool32 shaderUniformTexelBufferArrayDynamicIndexing;
    VkBool32 shaderStorageTexelBufferArrayDynamicIndexing;
    VkBool32 shaderUniformBufferArrayNonUniformIndexing;
    VkBool32 shaderSampledImageArrayNonUniformIndexing;
    VkBool32 shaderStorageBufferArrayNonUniformIndexing;
    VkBool32 shaderStorageImageArrayNonUniformIndexing;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexing;
    VkBool32 shaderUniformTexelBufferArrayNonUniformIndexing;
    VkBool32 shaderStorageTexelBufferArrayNonUniformIndexing;
    VkBool32 descriptorBindingUniformBufferUpdateAfterBind;
    VkBool32 descriptorBindingSampledImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageImageUpdateAfterBind;
    VkBool32 descriptorBindingStorageBufferUpdateAfterBind;
    VkBool32 descriptorBindingUniformTexelBufferUpdateAfterBind;
    VkBool32 descriptorBindingStorageTexelBufferUpdateAfterBind;
    VkBool32 descriptorBindingUpdateUnusedWhilePending;
    VkBool32 descriptorBindingPartiallyBound;
    VkBool32 descriptorBindingVariableDescriptorCount;
    VkBool32 runtimeDescriptorArray;
} VkPhysicalDeviceDescriptorIndexingFeatures;

or the equivalent

// Provided by VK_EXT_descriptor_indexing
typedef VkPhysicalDeviceDescriptorIndexingFeatures VkPhysicalDeviceDescriptorIndexingFeaturesEXT;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **shaderInputAttachmentArrayDynamicIndexing** indicates whether arrays of input attachments can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT** must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the **InputAttachmentArrayDynamicIndexing** capability.

- **shaderUniformTexelBufferArrayDynamicIndexing** indicates whether arrays of uniform texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of **VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER** must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can...
declare the `UniformTexelBufferArrayDynamicIndexing` capability.

- **shaderStorageTexelBufferArrayDynamicIndexing** indicates whether arrays of storage texel buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageTexelBufferArrayDynamicIndexing` capability.

- **shaderUniformBufferArrayNonUniformIndexing** indicates whether arrays of uniform buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `UniformBufferArrayNonUniformIndexing` capability.

- **shaderSampledImageArrayNonUniformIndexing** indicates whether arrays of samplers or sampled images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `SampledImageArrayNonUniformIndexing` capability.

- **shaderStorageBufferArrayNonUniformIndexing** indicates whether arrays of storage buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageBufferArrayNonUniformIndexing` capability.

- **shaderStorageImageArrayNonUniformIndexing** indicates whether arrays of storage images can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageImageArrayNonUniformIndexing` capability.

- **shaderInputAttachmentArrayNonUniformIndexing** indicates whether arrays of input attachments can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `InputAttachmentArrayNonUniformIndexing` capability.

- **shaderUniformTexelBufferArrayNonUniformIndexing** indicates whether arrays of uniform texel buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `UniformTexelBufferArrayNonUniformIndexing` capability.

- **shaderStorageTexelBufferArrayNonUniformIndexing** indicates whether arrays of storage texel
buffers can be indexed by non-uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` must not be indexed by non-uniform integer expressions when aggregated into arrays in shader code. This also indicates whether shader modules can declare the `StorageTexelBufferArrayNonUniformIndexing` capability.

- `descriptorBindingUniformBufferUpdateAfterBind` indicates whether the implementation supports updating uniform buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`.

- `descriptorBindingSampledImageUpdateAfterBind` indicates whether the implementation supports updating sampled image descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_SAMPLER`, `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, or `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`.

- `descriptorBindingStorageImageUpdateAfterBind` indicates whether the implementation supports updating storage image descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`.

- `descriptorBindingStorageBufferUpdateAfterBind` indicates whether the implementation supports updating storage buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`.

- `descriptorBindingUniformTexelBufferUpdateAfterBind` indicates whether the implementation supports updating uniform texel buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`.

- `descriptorBindingStorageTexelBufferUpdateAfterBind` indicates whether the implementation supports updating storage texel buffer descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`.

- `descriptorBindingUpdateUnusedWhilePending` indicates whether the implementation supports updating descriptors while the set is in use. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT` must not be used.

- `descriptorBindingPartiallyBound` indicates whether the implementation supports statically using a descriptor set binding in which some descriptors are not valid. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT` must not be used.

- `descriptorBindingVariableDescriptorCount` indicates whether the implementation supports descriptor sets with a variable-sized last binding. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT` must not be used.

- `runtimeDescriptorArray` indicates whether the implementation supports the SPIR-V `RuntimeDescriptorArray` capability. If this feature is not enabled, descriptors must not be declared in runtime arrays.

If the `VkPhysicalDeviceDescriptorIndexingFeatures` structure is included in the `pNext` chain of the
VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceDescriptorIndexingFeatures can also be used in thepNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDescriptorIndexingFeatures-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES

The VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_vertex_attribute_divisor
typedef struct VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           vertexAttributeInstanceRateDivisor;
    VkBool32           vertexAttributeInstanceRateZeroDivisor;
} VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **vertexAttributeInstanceRateDivisor** specifies whether vertex attribute fetching may be repeated in case of instanced rendering.
- **vertexAttributeInstanceRateZeroDivisor** specifies whether a zero value for VkVertexInputBindingDivisorDescriptionEXT::divisor is supported.

If the VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT

The VkPhysicalDeviceASTCDecodeFeaturesEXT structure is defined as:
typedef struct VkPhysicalDeviceASTCDecodeFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           decodeModeSharedExponent;
} VkPhysicalDeviceASTCDecodeFeaturesEXT;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **decodeModeSharedExponent** indicates whether the implementation supports decoding ASTC compressed formats to `VK_FORMAT_E5B9G9R9_UFLOAT_PACK32` internal precision.

If the `VkPhysicalDeviceASTCDecodeFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceASTCDecodeFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceASTCDecodeFeaturesEXT-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT`

The `VkPhysicalDeviceTransformFeedbackFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_transform_feedback
typedef struct VkPhysicalDeviceTransformFeedbackFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           transformFeedback;
    VkBool32           geometryStreams;
} VkPhysicalDeviceTransformFeedbackFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **transformFeedback** indicates whether the implementation supports transform feedback and shader modules can declare the `TransformFeedback` capability.
- **geometryStreams** indicates whether the implementation supports the `GeometryStreams` SPIR-V capability.

If the `VkPhysicalDeviceTransformFeedbackFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to
indicate whether each corresponding feature is supported. \( \text{VkPhysicalDeviceTransformFeedbackFeaturesEXT} \) can also be used in the \( \text{pNext} \) chain of \( \text{VkDeviceCreateInfo} \) to selectively enable these features.

### Valid Usage (Implicit)

- VUID-\( \text{VkPhysicalDeviceTransformFeedbackFeaturesEXT} \)-sType-sType
  - \( \text{sType} \) must be \( \text{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_FEATURES_EXT} \)

The \( \text{VkPhysicalDeviceVulkanMemoryModelFeatures} \) structure is defined as:

```c
typedef struct VkPhysicalDeviceVulkanMemoryModelFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           vulkanMemoryModel;
    VkBool32           vulkanMemoryModelDeviceScope;
    VkBool32           vulkanMemoryModelAvailabilityVisibilityChains;
} VkPhysicalDeviceVulkanMemoryModelFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_vulkan_memory_model
typedef VkPhysicalDeviceVulkanMemoryModelFeatures
        VkPhysicalDeviceVulkanMemoryModelFeaturesKHR;
```

This structure describes the following features:

- \( \text{sType} \) is the type of this structure.
- \( \text{pNext} \) is NULL or a pointer to a structure extending this structure.
- \( \text{vulkanMemoryModel} \) indicates whether the Vulkan Memory Model is supported, as defined in Vulkan Memory Model. This also indicates whether shader modules can declare the VulkanMemoryModel capability.
- \( \text{vulkanMemoryModelDeviceScope} \) indicates whether the Vulkan Memory Model can use Device scope synchronization. This also indicates whether shader modules can declare the VulkanMemoryModelDeviceScope capability.
- \( \text{vulkanMemoryModelAvailabilityVisibilityChains} \) indicates whether the Vulkan Memory Model can use availability and visibility chains with more than one element.

If the \( \text{VkPhysicalDeviceVulkanMemoryModelFeaturesKHR} \) structure is included in the \( \text{pNext} \) chain of the \( \text{VkPhysicalDeviceFeatures2} \) structure passed to \( \text{vkGetPhysicalDeviceFeatures2} \), it is filled in to indicate whether each corresponding feature is supported. \( \text{VkPhysicalDeviceVulkanMemoryModelFeaturesKHR} \) can also be used in the \( \text{pNext} \) chain of \( \text{VkDeviceCreateInfo} \) to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceVulkanMemoryModelFeatures-sType-sType
  
sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES

The VkPhysicalDeviceInlineUniformBlockFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_inline_uniform_block
typedef struct VkPhysicalDeviceInlineUniformBlockFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           inlineUniformBlock;
    VkBool32           descriptorBindingInlineUniformBlockUpdateAfterBind;
} VkPhysicalDeviceInlineUniformBlockFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **inlineUniformBlock** indicates whether the implementation supports inline uniform block descriptors. If this feature is not enabled, **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT** must not be used.
- **descriptorBindingInlineUniformBlockUpdateAfterBind** indicates whether the implementation supports updating inline uniform block descriptors after a set is bound. If this feature is not enabled, **VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT** must not be used with **VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT**.

If the VkPhysicalDeviceInlineUniformBlockFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceInlineUniformBlockFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceInlineUniformBlockFeaturesEXT-sType-sType
  
sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES_EXT

The VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV structure is defined as:
typedef struct VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV {
    VkStructureType      sType;
    void*                pNext;
    VkBool32             representativeFragmentTest;
} VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **representativeFragmentTest** indicates whether the implementation supports the representative fragment test. See Representative Fragment Test.

If the `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

**Valid Usage (Implicit)**

- [VUID-VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV-sType-sType] `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_REPRESENTATIVE_FRAGMENT_TEST_FEATURES_NV`.

---

The `VkPhysicalDeviceExclusiveScissorFeaturesNV` structure is defined as:

typedef struct VkPhysicalDeviceExclusiveScissorFeaturesNV {
    VkStructureType      sType;
    void*                pNext;
    VkBool32             exclusiveScissor;
} VkPhysicalDeviceExclusiveScissorFeaturesNV;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **exclusiveScissor** indicates that the implementation supports the exclusive scissor test.

See Exclusive Scissor Test for more information.

If the `VkPhysicalDeviceExclusiveScissorFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to
indicate whether each corresponding feature is supported. 

\texttt{VkPhysicalDeviceExclusiveScissorFeaturesNV} can also be used in the \texttt{pNext} chain of \texttt{VkDeviceCreateInfo} to selectively enable these features.

### Valid Usage (Implicit)

- \textit{VUID-VkPhysicalDeviceExclusiveScissorFeaturesNV-sType-sType}
  
  \textit{sType} must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXCLUSIVE_SCISSOR_FEATURES_NV}

The \texttt{VkPhysicalDeviceCornerSampledImageFeaturesNV} structure is defined as:

```c
// Provided by VK_NV_corner_sampled_image
typedef struct VkPhysicalDeviceCornerSampledImageFeaturesNV {
    VkStructureType sType;
    void* pNext;
    VkBool32 cornerSampledImage;
} VkPhysicalDeviceCornerSampledImageFeaturesNV;
```

This structure describes the following feature:

- \textit{sType} is the type of this structure.
- \textit{pNext} is NULL or a pointer to a structure extending this structure.
- \textit{cornerSampledImage} specifies whether images can be created with a \texttt{VkImageCreateInfo::flags} containing \texttt{VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV}. See Corner-Sampled Images.

If the \texttt{VkPhysicalDeviceCornerSampledImageFeaturesNV} structure is included in the \texttt{pNext} chain of the \texttt{VkPhysicalDeviceFeatures2} structure passed to \texttt{vkGetPhysicalDeviceFeatures2}, it is filled in to indicate whether each corresponding feature is supported. \texttt{VkPhysicalDeviceCornerSampledImageFeaturesNV} can also be used in the \texttt{pNext} chain of \texttt{VkDeviceCreateInfo} to selectively enable these features.

### Valid Usage (Implicit)

- \textit{VUID-VkPhysicalDeviceCornerSampledImageFeaturesNV-sType-sType}
  
  \textit{sType} must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CORNER_SAMPLED_IMAGE_FEATURES_NV}

The \texttt{VkPhysicalDeviceComputeShaderDerivativesFeaturesNV} structure is defined as:
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **computeDerivativeGroupQuads** indicates that the implementation supports the `ComputeDerivativeGroupQuadsNV` SPIR-V capability.
- **computeDerivativeGroupLinear** indicates that the implementation supports the `ComputeDerivativeGroupLinearNV` SPIR-V capability.

See **Quad shader scope** for more information.

If the `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceComputeShaderDerivativesFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceComputeShaderDerivativesFeaturesNV-sType-sType

The `VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV` structure is defined as:

```c
typedef struct VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           fragmentShaderBarycentric;
} VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **fragmentShaderBarycentric** indicates that the implementation supports the `BaryCoordNV` and
BaryCoordNoPerspNV SPIR-V fragment shader built-ins and supports the PerVertexNV SPIR-V decoration on fragment shader input variables.

See Barycentric Interpolation for more information.

If the VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV-sType-sType
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_BARYCENTRIC_FEATURES_NV

The VkPhysicalDeviceShaderImageFootprintFeaturesNV structure is defined as:

```c
// Provided by VK_NV_shader_image_footprint
typedef struct VkPhysicalDeviceShaderImageFootprintFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           imageFootprint;
} VkPhysicalDeviceShaderImageFootprintFeaturesNV;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **imageFootprint** specifies whether the implementation supports the ImageFootprintNV SPIR-V capability.

See Texel Footprint Evaluation for more information.

If the VkPhysicalDeviceShaderImageFootprintFeaturesNV structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceShaderImageFootprintFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderImageFootprintFeaturesNV-sType-sType
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_FOOTPRINT_FEATURES_NV
The `VkPhysicalDeviceShadingRateImageFeaturesNV` structure is defined as:

```c
// Provided by VK_NV_shading_rate_image
typedef struct VkPhysicalDeviceShadingRateImageFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shadingRateImage;
    VkBool32           shadingRateCoarseSampleOrder;
} VkPhysicalDeviceShadingRateImageFeaturesNV;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shadingRateImage** indicates that the implementation supports the use of a shading rate image to derive an effective shading rate for fragment processing. It also indicates that the implementation supports the `ShadingRateNV` SPIR-V execution mode.
- **shadingRateCoarseSampleOrder** indicates that the implementation supports a user-configurable ordering of coverage samples in fragments larger than one pixel.

See [Shading Rate Image](#) for more information.

If the `VkPhysicalDeviceShadingRateImageFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShadingRateImageFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

**Valid Usage (Implicit)**

```c
• VUID-VkPhysicalDeviceShadingRateImageFeaturesNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_FEATURES_NV
```

The `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_fragment_density_map
typedef struct VkPhysicalDeviceFragmentDensityMapFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           fragmentDensityMap;
    VkBool32           fragmentDensityMapDynamic;
    VkBool32           fragmentDensityMapNonSubsampledImages;
} VkPhysicalDeviceFragmentDensityMapFeaturesEXT;
```

This structure describes the following features:
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **fragmentDensityMap** specifies whether the implementation supports render passes with a fragment density map attachment. If this feature is not enabled and the **pNext** chain of `VkRenderPassCreateInfo` includes a `VkRenderPassFragmentDensityMapCreateInfoEXT` structure, **fragmentDensityMapAttachment must be** **VK_ATTACHMENT_UNUSED**.

• **fragmentDensityMapDynamic** specifies whether the implementation supports dynamic fragment density map image views. If this feature is not enabled, `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT` must not be included in `VkImageViewCreateInfo::flags`.

• **fragmentDensityMapNonSubsampledImages** specifies whether the implementation supports regular non-subsampled image attachments with fragment density map render passes. If this feature is not enabled, render passes with a fragment density map attachment must only have subsampled attachments bound.

If the `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` structure is included in the **pNext** chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceFragmentDensityMapFeaturesEXT` can also be used in the **pNext** chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceFragmentDensityMapFeaturesEXT-sType-sType**

  - **sType must be** **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_FEATURES_EXT**

The `VkPhysicalDeviceFragmentDensityMap2FeaturesEXT` structure is defined as:

```c
typedef struct VkPhysicalDeviceFragmentDensityMap2FeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           fragmentDensityMapDeferred;
} VkPhysicalDeviceFragmentDensityMap2FeaturesEXT;
```

This structure describes the following features:

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **fragmentDensityMapDeferred** specifies whether the implementation supports deferred reads of fragment density map image views. If this feature is not enabled, `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DEFERRED_BIT_EXT` must not be included in `VkImageViewCreateInfo::flags`.

If the `VkPhysicalDeviceFragmentDensityMap2FeaturesEXT` structure is included in the **pNext** chain of
the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceFragmentDensityMap2FeaturesEXT can also be used in thepNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceFragmentDensityMap2FeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_2_FEATURES_EXT

The VkPhysicalDeviceScalarBlockLayoutFeatures structure is defined as:

```c
typedef struct VkPhysicalDeviceScalarBlockLayoutFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           scalarBlockLayout;
} VkPhysicalDeviceScalarBlockLayoutFeatures;
```

or the equivalent

```c
// Provided by VK_EXT_scalar_block_layout
typedef VkPhysicalDeviceScalarBlockLayoutFeatures
VkPhysicalDeviceScalarBlockLayoutFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **scalarBlockLayout** indicates that the implementation supports the layout of resource blocks in shaders using scalar alignment.

If the VkPhysicalDeviceScalarBlockLayoutFeatures structure is included in thepNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceScalarBlockLayoutFeatures can also be used in thepNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceScalarBlockLayoutFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES

The VkPhysicalDeviceUniformBufferStandardLayoutFeatures structure is defined as:
typedef struct VkPhysicalDeviceUniformBufferStandardLayoutFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 uniformBufferStandardLayout;
} VkPhysicalDeviceUniformBufferStandardLayoutFeatures;

or the equivalent

// Provided by VK_KHR_uniform_buffer_standard_layout
typedef VkPhysicalDeviceUniformBufferStandardLayoutFeatures
VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure. 
- **uniformBufferStandardLayout** indicates that the implementation supports the same layouts for uniform buffers as for storage and other kinds of buffers. See Standard Buffer Layout.

If the VkPhysicalDeviceUniformBufferStandardLayoutFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceUniformBufferStandardLayoutFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceUniformBufferStandardLayoutFeatures-sType-sType

sType must be

VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT_FEATURES

The VkPhysicalDeviceDepthClipEnableFeaturesEXT structure is defined as:

// Provided by VK_EXT_depth_clip_enable
typedef struct VkPhysicalDeviceDepthClipEnableFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 depthClipEnable;
} VkPhysicalDeviceDepthClipEnableFeaturesEXT;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
• `depthClipEnable` indicates that the implementation supports setting the depth clipping operation explicitly via the `VkPipelineRasterizationDepthClipStateCreateInfoEXT` pipeline state. Otherwise, depth clipping is only enabled when `VkPipelineRasterizationStateCreateInfo::depthClampEnable` is set to `VK_FALSE`.

If the `VkPhysicalDeviceDepthClipEnableFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDepthClipEnableFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

**VUID-VkPhysicalDeviceDepthClipEnableFeaturesEXT-sType-sType**

`sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLE_FEATURES_EXT`

The `VkPhysicalDeviceMemoryPriorityFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_memory_priority
typedef struct VkPhysicalDeviceMemoryPriorityFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           memoryPriority;
} VkPhysicalDeviceMemoryPriorityFeaturesEXT;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `memoryPriority` indicates that the implementation supports memory priorities specified at memory allocation time via `VkMemoryPriorityAllocateInfoEXT`.

If the `VkPhysicalDeviceMemoryPriorityFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceMemoryPriorityFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

**VUID-VkPhysicalDeviceMemoryPriorityFeaturesEXT-sType-sType**

`sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PRIORITY_FEATURES_EXT`

The `VkPhysicalDeviceBufferDeviceAddressFeatures` structure is defined as:
typedef struct VkPhysicalDeviceBufferDeviceAddressFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           bufferDeviceAddress;
    VkBool32           bufferDeviceAddressCaptureReplay;
    VkBool32           bufferDeviceAddressMultiDevice;
} VkPhysicalDeviceBufferDeviceAddressFeatures;

or the equivalent

// Provided by VK_KHR_buffer_device_address
typedef VkPhysicalDeviceBufferDeviceAddressFeatures
    VkPhysicalDeviceBufferDeviceAddressFeaturesKHR;

This structure describes the following features:

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **bufferDeviceAddress** indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from vkGetBufferDeviceAddress.
• **bufferDeviceAddressCaptureReplay** indicates that the implementation supports saving and reusing buffer and device addresses, e.g. for trace capture and replay.
• **bufferDeviceAddressMultiDevice** indicates that the implementation supports the bufferDeviceAddress, rayTracingPipeline and rayQuery features for logical devices created with multiple physical devices. If this feature is not supported, buffer and acceleration structure addresses must not be queried on a logical device created with more than one physical device.

**Note**
bufferDeviceAddressMultiDevice exists to allow certain legacy platforms to be able to support bufferDeviceAddress without needing to support shared GPU virtual addresses for multi-device configurations.

See vkGetBufferDeviceAddress for more information.

If the VkPhysicalDeviceBufferDeviceAddressFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceBufferDeviceAddressFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

**Valid Usage (Implicit)**

• VUID-VkPhysicalDeviceBufferDeviceAddressFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES
The `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_buffer_device_address
typedef struct VkPhysicalDeviceBufferDeviceAddressFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           bufferDeviceAddress;
    VkBool32           bufferDeviceAddressCaptureReplay;
    VkBool32           bufferDeviceAddressMultiDevice;
} VkPhysicalDeviceBufferDeviceAddressFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **bufferDeviceAddress** indicates that the implementation supports accessing buffer memory in shaders as storage buffers via an address queried from `vkGetBufferDeviceAddressEXT`.
- **bufferDeviceAddressCaptureReplay** indicates that the implementation supports saving and reusing buffer addresses, e.g. for trace capture and replay.
- **bufferDeviceAddressMultiDevice** indicates that the implementation supports the `bufferDeviceAddress` feature for logical devices created with multiple physical devices. If this feature is not supported, buffer addresses must not be queried on a logical device created with more than one physical device.

If the `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Note**

The `VkPhysicalDeviceBufferDeviceAddressFeaturesEXT` structure has the same members as the `VkPhysicalDeviceBufferDeviceAddressFeatures` structure, but the functionality indicated by the members is expressed differently. The features indicated by the `VkPhysicalDeviceBufferDeviceAddressFeatures` structure requires additional flags to be passed at memory allocation time, and the capture and replay mechanism is built around opaque capture addresses for buffer and memory objects.

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceBufferDeviceAddressFeaturesEXT-sType-sType**
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT`

The `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` structure is defined as:
typedef struct VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           dedicatedAllocationImageAliasing;
} VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV;

This structure describes the following feature:

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **dedicatedAllocationImageAliasing** indicates that the implementation supports aliasing of compatible image objects on a dedicated allocation.

If the `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEDICATED_ALLOCATION_IMAGE_ALIASING_FEATURES_NV`

The `VkPhysicalDeviceImagelessFramebufferFeatures` structure is defined as:

```c
typedef struct VkPhysicalDeviceImagelessFramebufferFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           imagelessFramebuffer;
} VkPhysicalDeviceImagelessFramebufferFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_imageless_framebuffer
typedef VkPhysicalDeviceImagelessFramebufferFeatures
    VkPhysicalDeviceImagelessFramebufferFeaturesKHR;
```

This structure describes the following feature:

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.
• **imagelessFramebuffer** indicates that the implementation supports specifying the image view for attachments at render pass begin time via **VkRenderPassAttachmentBeginInfo**.

If the **VkPhysicalDeviceImagelessFramebufferFeatures** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceImagelessFramebufferFeatures** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceImagelessFramebufferFeatures-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES**

The **VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT** structure is defined as:

```c
// Provided by VK_EXT_fragment_shader_interlock
typedef struct VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           fragmentShaderSampleInterlock;
    VkBool32           fragmentShaderPixelInterlock;
    VkBool32           fragmentShaderShadingRateInterlock;
} VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **fragmentShaderSampleInterlock** indicates that the implementation supports the **FragmentShaderSampleInterlockEXT** SPIR-V capability.
- **fragmentShaderPixelInterlock** indicates that the implementation supports the **FragmentShaderPixelInterlockEXT** SPIR-V capability.
- **fragmentShaderShadingRateInterlock** indicates that the implementation supports the **FragmentShaderShadingRateInterlockEXT** SPIR-V capability.

If the **VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT

The `VkPhysicalDeviceCooperativeMatrixFeaturesNV` structure is defined as:

```c
// Provided by VK_NV_cooperative_matrix
typedef struct VkPhysicalDeviceCooperativeMatrixFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           cooperativeMatrix;
    VkBool32           cooperativeMatrixRobustBufferAccess;
} VkPhysicalDeviceCooperativeMatrixFeaturesNV;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `cooperativeMatrix` indicates that the implementation supports the `CooperativeMatrixNV SPIR-V` capability.
- `cooperativeMatrixRobustBufferAccess` indicates that the implementation supports robust buffer access for SPIR-V `OpCooperativeMatrixLoadNV` and `OpCooperativeMatrixStoreNV` instructions.

If the `VkPhysicalDeviceCooperativeMatrixFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceCooperativeMatrixFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceCooperativeMatrixFeaturesNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_FEATURES_NV

The `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_ycbcr_image_arrays
typedef struct VkPhysicalDeviceYcbcrImageArraysFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           ycbcrImageArrays;
} VkPhysicalDeviceYcbcrImageArraysFeaturesEXT;
```
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **ycbcrImageArrays** indicates that the implementation supports creating images with a format that requires Y'CbCr conversion and has multiple array layers.

If the `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceYcbcrImageArraysFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceYcbcrImageArraysFeaturesEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT`.

The `VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures` structure is defined as:

```c
typedef struct VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderSubgroupExtendedTypes;
} VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_shader_subgroup_extended_types
typedef VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures
VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderSubgroupExtendedTypes** is a boolean that specifies whether subgroup operations can use 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types in group operations with subgroup scope, if the implementation supports the types.

If the `VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderSubgroupExtendedTypesFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
The `VkPhysicalDeviceHostQueryResetFeatures` structure is defined as:

```c
typedef struct VkPhysicalDeviceHostQueryResetFeatures {
    VkStructureType sType;
    void* pNext;
    VkBool32 hostQueryReset;
} VkPhysicalDeviceHostQueryResetFeatures;
```

or the equivalent

```c
// Provided by VK_EXT_host_query_reset
typedef VkPhysicalDeviceHostQueryResetFeatures
VkPhysicalDeviceHostQueryResetFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **hostQueryReset** indicates that the implementation supports resetting queries from the host with `vkResetQueryPool`.

If the `VkPhysicalDeviceHostQueryResetFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceHostQueryResetFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

The `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL` structure is defined as:

```c
typedef VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL {
    VkPhysicalDeviceFeatures2 hostFeatures;
    VkPhysicalDeviceHostQueryResetFeatures hostQueryResetFeatures;
} VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL;
```
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderIntegerFunctions2** indicates that the implementation supports the `IntegerFunctions2INTEL` SPIR-V capability.

If the `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL.features` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL-sType-sType

- sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_FUNCTIONS_2_FEATURES_INTEL`

The `VkPhysicalDeviceCoverageReductionModeFeaturesNV` structure is defined as:

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **coverageReductionMode** indicates whether the implementation supports coverage reduction modes. See Coverage Reduction.

If the `VkPhysicalDeviceCoverageReductionModeFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported.
VkPhysicalDeviceCoverageReductionModeFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceCoverageReductionModeFeaturesNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COVERAGE_REDUCTION_MODE_FEATURES_NV

The VkPhysicalDeviceTimelineSemaphoreFeatures structure is defined as:

```c
typedef struct VkPhysicalDeviceTimelineSemaphoreFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           timelineSemaphore;
} VkPhysicalDeviceTimelineSemaphoreFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_timeline_semaphore
typedef VkPhysicalDeviceTimelineSemaphoreFeatures
    VkPhysicalDeviceTimelineSemaphoreFeaturesKHR;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **timelineSemaphore** indicates whether semaphores created with a VkSemaphoreType of VK_SEMAPHORE_TYPE_TIMELINE are supported.

If the VkPhysicalDeviceTimelineSemaphoreFeatures structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceTimelineSemaphoreFeatures can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceTimelineSemaphoreFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES

The VkPhysicalDeviceIndexTypeUint8FeaturesEXT structure is defined as:
This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `indexTypeUint8` indicates that `VK_INDEX_TYPE_UINT8_EXT` can be used with `vkCmdBindIndexBuffer`.

If the `VkPhysicalDeviceIndexTypeUint8FeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceIndexTypeUint8FeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceIndexTypeUint8FeaturesEXT-sType-sType`  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL DEVICE_INDEX_TYPE_UINT8 FEATURES_EXT`

The `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` structure is defined as:

```c
// Provided by VK_NV_shader_sm_builtins
typedef struct VkPhysicalDeviceShaderSMBuiltinsFeaturesNV {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderSMBuiltins;
} VkPhysicalDeviceShaderSMBuiltinsFeaturesNV;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderSMBuiltins` indicates whether the implementation supports the SPIR-V `ShaderSMBuiltinsNV` capability.

If the `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderSMBuiltinsFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderSMBuiltinsFeaturesNV-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_FEATURES_NV

The `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures` structure is defined as:

```c
typedef struct VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           separateDepthStencilLayouts;
} VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures;
```

or the equivalent

```c
// Provided by VK_KHR_separate_depth_stencil_layouts
typedef VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures
    VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **separateDepthStencilLayouts** indicates whether the implementation supports a `VkImageMemoryBarrier` for a depth/stencil image with only one of `VK_IMAGE_ASPECT_DEPTH_BIT` or `VK_IMAGE_ASPECT_STENCIL_BIT` set, and whether `VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL`, `VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL`, `VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL`, or `VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL` can be used.

If the `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceSeparateDepthStencilLayoutsFeatures-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES

The `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` structure is defined as:
typedef struct VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 pipelineExecutableInfo;
} VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **pipelineExecutableInfo** indicates that the implementation supports reporting properties and statistics about the executables associated with a compiled pipeline.

If the `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

Valid Usage (Implicit)

- VUID-VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR-sType-sType
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_EXECUTABLE_PROPERTIES_FEATURES_KHR`.

The `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT` structure is defined as:

typedef struct VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 shaderDemoteToHelperInvocation;
} VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderDemoteToHelperInvocation** indicates whether the implementation supports the SPIR-V DemoteToHelperInvocationEXT capability.

If the `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported.
VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT

The VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_texel_buffer_alignment
typedef struct VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           texelBufferAlignment;
} VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **texelBufferAlignment** indicates whether the implementation uses more specific alignment requirements advertised in VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT rather than VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment.

If the VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT

The VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT structure is defined as:

```c
// Provided by VK_EXT_texture_compression_astc_hdr
typedef struct VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           textureCompressionASTC_HDR;
} VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT;
```
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **textureCompressionASTC_HDR** indicates whether all of the ASTC HDR compressed texture formats are supported. If this feature is enabled, then the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`, `VK_FORMAT_FEATURE_BLIT_SRC_BIT` and `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT` features must be supported in `optimalTilingFeatures` for the following formats:
  - `VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT`
  - `VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT`

To query for additional properties, or if the feature is not enabled, `vkGetPhysicalDeviceFormatProperties` and `vkGetPhysicalDeviceImageFormatProperties` can be used to check for supported properties of individual formats as normal.

If the `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT-sType-sType
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT`

The `VkPhysicalDeviceLineRasterizationFeaturesEXT` structure is defined as:
typedef struct VkPhysicalDeviceLineRasterizationFeaturesEXT {
    VkStructureType   sType;
    void*              pNext;
    VkBool32           rectangularLines;
    VkBool32           bresenhamLines;
    VkBool32           smoothLines;
    VkBool32           stippledRectangularLines;
    VkBool32           stippledBresenhamLines;
    VkBool32           stippledSmoothLines;
} VkPhysicalDeviceLineRasterizationFeaturesEXT;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **rectangularLines** indicates whether the implementation supports rectangular line rasterization.
- **bresenhamLines** indicates whether the implementation supports Bresenham-style line rasterization.
- **smoothLines** indicates whether the implementation supports smooth line rasterization.
- **stippledRectangularLines** indicates whether the implementation supports stippled line rasterization with VK_LINE_RASTERIZATION_MODE_RECTANGULAR_EXT lines, or with VK_LINE_RASTERIZATION_MODE_DEFAULT_EXT lines if VkPhysicalDeviceLimits::strictLines is VK_TRUE.
- **stippledBresenhamLines** indicates whether the implementation supports stippled line rasterization with VK_LINE_RASTERIZATION_MODE_BRESENHAM_EXT lines.
- **stippledSmoothLines** indicates whether the implementation supports stippled line rasterization with VK_LINE_RASTERIZATION_MODE_RECTANGULAR_SMOOTH_EXT lines.

If the VkPhysicalDeviceLineRasterizationFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceLineRasterizationFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceLineRasterizationFeaturesEXT-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT

The VkPhysicalDeviceSubgroupSizeControlFeaturesEXT structure is defined as:
typedef struct VkPhysicalDeviceSubgroupSizeControlFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 subgroupSizeControl;
    VkBool32 computeFullSubgroups;
} VkPhysicalDeviceSubgroupSizeControlFeaturesEXT;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **subgroupSizeControl** indicates whether the implementation supports controlling shader subgroup sizes via the VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT flag and the VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure.
- **computeFullSubgroups** indicates whether the implementation supports requiring full subgroups in compute shaders via the VK_PIPELINE_SHADER_STAGE_CREATE_REQUIRE_FULL_SUBGROUPS_BIT_EXT flag.

If the VkPhysicalDeviceSubgroupSizeControlFeaturesEXT structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceSubgroupSizeControlFeaturesEXT can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

**Note**

The VkPhysicalDeviceSubgroupSizeControlFeaturesEXT structure was added in version 2 of the VK_EXT_subgroup_size_control extension. Version 1 implementations of this extension will not fill out the features structure but applications may assume that both subgroupSizeControl and computeFullSubgroups are supported if the extension is supported. (See also the Feature Requirements section.) Applications are advised to add a VkPhysicalDeviceSubgroupSizeControlFeaturesEXT structure to the pNext chain of VkDeviceCreateInfo to enable the features regardless of the version of the extension supported by the implementation. If the implementation only supports version 1, it will safely ignore the VkPhysicalDeviceSubgroupSizeControlFeaturesEXT structure.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceSubgroupSizeControlFeaturesEXT-sType-sType

  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT

The VkPhysicalDeviceCoherentMemoryFeaturesAMD structure is defined as:

2158 | Chapter 41. Features
typedef struct VkPhysicalDeviceCoherentMemoryFeaturesAMD {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           deviceCoherentMemory;
} VkPhysicalDeviceCoherentMemoryFeaturesAMD;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceCoherentMemory** indicates that the implementation supports device coherent memory.

If the `VkPhysicalDeviceCoherentMemoryFeaturesAMD` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceCoherentMemoryFeaturesAMD` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceCoherentMemoryFeaturesAMD-sType-sType
  
sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COHERENT_MEMORY_FEATURES_AMD

The `VkPhysicalDeviceAccelerationStructureFeaturesKHR` structure is defined as:

typedef struct VkPhysicalDeviceAccelerationStructureFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           accelerationStructure;
    VkBool32           accelerationStructureCaptureReplay;
    VkBool32           accelerationStructureIndirectBuild;
    VkBool32           accelerationStructureHostCommands;
    VkBool32           descriptorBindingAccelerationStructureUpdateAfterBind;
} VkPhysicalDeviceAccelerationStructureFeaturesKHR;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **accelerationStructure** indicates whether the implementation supports the acceleration structure functionality. See Acceleration Structures.
- **accelerationStructureCaptureReplay** indicates whether the implementation supports saving and reusing acceleration structure device addresses, e.g. for trace capture and replay.
• `accelerationStructureIndirectBuild` indicates whether the implementation supports indirect acceleration structure build commands, e.g. `vkCmdBuildAccelerationStructuresIndirectKHR`.

• `accelerationStructureHostCommands` indicates whether the implementation supports host side acceleration structure commands, e.g. `vkBuildAccelerationStructuresKHR`, `vkCopyAccelerationStructureKHR`, `vkCopyAccelerationStructureToMemoryKHR`, `vkCopyMemoryToAccelerationStructureKHR`, `vkWriteAccelerationStructuresPropertiesKHR`.

• `descriptorBindingAccelerationStructureUpdateAfterBind` indicates whether the implementation supports updating acceleration structure descriptors after a set is bound. If this feature is not enabled, `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT` must not be used with `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR`.

If the `VkPhysicalDeviceAccelerationStructureFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceAccelerationStructureFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceAccelerationStructureFeaturesKHR-sType-sType` 
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR`.

The `VkPhysicalDeviceRayTracingPipelineFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_ray_tracing_pipeline
typedef struct VkPhysicalDeviceRayTracingPipelineFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           rayTracingPipeline;
    VkBool32           rayTracingPipelineShaderGroupHandleCaptureReplay;
    VkBool32           rayTracingPipelineShaderGroupHandleCaptureReplayMixed;
    VkBool32           rayTracingPipelineTraceRaysIndirect;
    VkBool32           rayTraversalPrimitiveCulling;
} VkPhysicalDeviceRayTracingPipelineFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.

- `pNext` is `NULL` or a pointer to a structure extending this structure.

- `rayTracingPipeline` indicates whether the implementation supports the ray tracing pipeline functionality. See Ray Tracing.

- `rayTracingPipelineShaderGroupHandleCaptureReplay` indicates whether the implementation supports saving and reusing shader group handles, e.g. for trace capture and replay.

- `rayTraversalPrimitiveCulling` indicates whether the implementation supports reuse of shader group handles being arbitrarily mixed with creation of non-reused
shader group handles. If this is VK_FALSE, all reused shader group handles **must** be specified before any non-reused handles **may** be created.

- **rayTracingPipelineTraceRaysIndirect** indicates whether the implementation supports indirect trace ray commands, e.g. `vkCmdTraceRaysIndirectKHR`.
- **rayTraversalPrimitiveCulling** indicates whether the implementation supports **primitive culling** during ray traversal.

If the `VkPhysicalDeviceRayTracingPipelineFeaturesKHR` structure is included in the **pNext** chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceRayTracingPipelineFeaturesKHR` can also be used in the **pNext** chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage

- **VUID-VkPhysicalDeviceRayTracingPipelineFeaturesKHR-rayTracingPipelineShaderGroupHandleCaptureReplayMixed-03575**

  If `rayTracingPipelineShaderGroupHandleCaptureReplayMixed` is VK_TRUE, `rayTracingPipelineShaderGroupHandleCaptureReplay` must also be VK_TRUE

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceRayTracingPipelineFeaturesKHR-sType-sType**

  **sType** **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TrACING_PIPELINE_FEATURES_KHR

The `VkPhysicalDeviceRayQueryFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_ray_query
typedef struct VkPhysicalDeviceRayQueryFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    Vk Bool32 rayQuery;
} VkPhysicalDeviceRayQueryFeaturesKHR;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **rayQuery** indicates whether the implementation supports ray query (OpRayQueryProceedKHR) functionality.

If the `VkPhysicalDeviceRayQueryFeaturesKHR` structure is included in the **pNext** chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceRayQueryFeaturesKHR`
can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceRayQueryFeaturesKHR-sType-sType`  
  `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_QUERY_FEATURES_KHR`

The `VkPhysicalDeviceExtendedDynamicStateFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_extended_dynamic_state
typedef struct VkPhysicalDeviceExtendedDynamicStateFeaturesEXT {
    VkStructureType sType;  
    void* pNext;           
    VkBool32 extendedDynamicState;  
} VkPhysicalDeviceExtendedDynamicStateFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **extendedDynamicState** indicates that the implementation supports the following dynamic states:
  - `VK_DYNAMIC_STATE_CULL_MODE_EXT`
  - `VK_DYNAMIC_STATE_FRONT_FACE_EXT`
  - `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT`
  - `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT`
  - `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT`
  - `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_STENCIL_OP_EXT`

If the `VkPhysicalDeviceExtendedDynamicStateFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceExtendedDynamicStateFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceExtendedDynamicStateFeaturesEXT-sType-sType**
  
  *sType* must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_FEATURES_EXT**

The *VkPhysicalDeviceExtendedDynamicState2FeaturesEXT* structure is defined as:

```c
// Provided by VK_EXT_extended_dynamic_state2
typedef struct VkPhysicalDeviceExtendedDynamicState2FeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 extendedDynamicState2;
    VkBool32 extendedDynamicState2LogicOp;
    VkBool32 extendedDynamicState2PatchControlPoints;
} VkPhysicalDeviceExtendedDynamicState2FeaturesEXT;
```

The members of the *VkPhysicalDeviceExtendedDynamicState2FeaturesEXT* structure describe the following features:

- **extendedDynamicState2** indicates that the implementation supports the following dynamic states:
  
  - **VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT**
  - **VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT**
  - **VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT**

- **extendedDynamicState2LogicOp** indicates that the implementation supports the following dynamic state:
  
  - **VK_DYNAMIC_STATE_LOGIC_OP_EXT**

- **extendedDynamicState2PatchControlPoints** indicates that the implementation supports the following dynamic state:
  
  - **VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT**

If the *VkPhysicalDeviceExtendedDynamicState2FeaturesEXT* structure is included in the *pNext* chain of *VkPhysicalDeviceFeatures2*, it is filled with values indicating whether the feature is supported. *VkPhysicalDeviceExtendedDynamicState2FeaturesEXT* can also be used in the *pNext* chain of *VkDeviceCreateInfo* to enable features.

Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV-sType-sType**
  
  *sType* must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_FEATURES_NV**

The *VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV* structure is defined as:
typedef struct VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV {
    VkStructureType sType;
    void* pNext;
    VkBool32 deviceGeneratedCommands;
} VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceGeneratedCommands** indicates whether the implementation supports functionality to generate commands on the device. See Device-Generated Commands.

If the `VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_FEATURES_NV`

The `VkPhysicalDeviceDiagnosticsConfigFeaturesNV` structure is defined as:

```c
// Provided by VK_NV_device_diagnostics_config
typedef struct VkPhysicalDeviceDiagnosticsConfigFeaturesNV {
    VkStructureType sType;
    void* pNext;
    VkBool32 diagnosticsConfig;
} VkPhysicalDeviceDiagnosticsConfigFeaturesNV;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **diagnosticsConfig** indicates whether the implementation supports the ability to configure diagnostic tools.

If the `VkPhysicalDeviceDiagnosticsConfigFeaturesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceDiagnosticsConfigFeaturesNV` can also be used in the `pNext` chain of...
**VkDeviceCreateInfo** to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDiagnosticsConfigFeaturesNV-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DIAGNOSTICS_CONFIG_FEATURES_NV`

The **VkPhysicalDeviceDeviceMemoryReportFeaturesEXT** structure is defined as:

```c
typedef struct VkPhysicalDeviceDeviceMemoryReportFeaturesEXT {
  VkStructureType    sType;
  void*              pNext;
  VkBool32           deviceMemoryReport;
} VkPhysicalDeviceDeviceMemoryReportFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **deviceMemoryReport** indicates whether the implementation supports the ability to register device memory report callbacks.

If the **VkPhysicalDeviceDeviceMemoryReportFeaturesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceDeviceMemoryReportFeaturesEXT** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDeviceMemoryReportFeaturesEXT-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_MEMORY_REPORT_FEATURES_EXT`

The **VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT** structure is defined as:

```c
typedef struct VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT {
  VkStructureType    sType;
  void*              pNext;
  VkBool32           pipelineCreationCacheControl;
} VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT;
```

This structure describes the following feature:
• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **pipelineCreationCacheControl** indicates that the implementation supports:
  - The following **can** be used in **Vk*PipelineCreateInfo::flags**:
    - `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT`
    - `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT`
  - The following **can** be used in **VkPipelineCacheCreateInfo::flags**:
    - `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT`

If the **VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT** **can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

---

### Valid Usage (Implicit)

- **VUID-VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT-sType-sType**

**sType** must be:

```plaintext
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_CREATION_CACHE_CONTROL_FEATURES_EXT
```

---

The **VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR** structure is defined as:

```c
// Provided by VK_KHR_zero_initialize_workgroup_memory
typedef struct VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderZeroInitializeWorkgroupMemory;
} VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR;
```

This structure describes the following feature:

• **sType** is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.
• **shaderZeroInitializeWorkgroupMemory** specifies whether the implementation supports initializing a variable in Workgroup storage class.

If the **VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR** **can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR-sType-sType
  sType must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ZERO_INITIALIZE_WORKGROUP_MEMORY_FEATURES_KHR}

The \texttt{VkPhysicalDevicePrivateDataFeaturesEXT} structure is defined as:

```c
// Provided by VK_EXT_private_data
typedef struct VkPhysicalDevicePrivateDataFeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 privateData;
} VkPhysicalDevicePrivateDataFeaturesEXT;
```

This structure describes the following feature:

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is NULL or a pointer to a structure extending this structure.
- \texttt{privateData} indicates whether the implementation supports private data. See Private Data.

If the \texttt{VkPhysicalDevicePrivateDataFeaturesEXT} structure is included in the \texttt{pNext} chain of the \texttt{VkPhysicalDeviceFeatures2} structure passed to \texttt{vkGetPhysicalDeviceFeatures2}, it is filled in to indicate whether each corresponding feature is supported. \texttt{VkPhysicalDevicePrivateDataFeaturesEXT} can also be used in the \texttt{pNext} chain of \texttt{VkDeviceCreateInfo} to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDevicePrivateDataFeaturesEXT-sType-sType
  sType must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PRIVATE_DATA_FEATURES_EXT}

The \texttt{VkPhysicalDeviceRobustness2FeaturesEXT} structure is defined as:

```c
// Provided by VK_EXT_robustness2
typedef struct VkPhysicalDeviceRobustness2FeaturesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 robustBufferAccess2;
    VkBool32 robustImageAccess2;
    VkBool32 nullDescriptor;
} VkPhysicalDeviceRobustness2FeaturesEXT;
```

This structure describes the following features:

- \texttt{sType} is the type of this structure.
• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **robustBufferAccess2** indicates whether buffer accesses are tightly bounds-checked against the range of the descriptor. Uniform buffers **must** be bounds-checked to the range of the descriptor, where the range is rounded up to a multiple of **robustUniformBufferAccessSizeAlignment**. Storage buffers **must** be bounds-checked to the range of the descriptor, where the range is rounded up to a multiple of **robustStorageBufferAccessSizeAlignment**. Out of bounds buffer loads will return zero values, and formatted loads will have (0,0,1) values inserted for missing G, B, or A components based on the format.

• **robustImageAccess2** indicates whether image accesses are tightly bounds-checked against the dimensions of the image view. Out of bounds image loads will return zero values, with (0,0,1) values inserted for missing G, B, or A components based on the format.

• **nullDescriptor** indicates whether descriptors **can** be written with a **VK_NULL_HANDLE** resource or view, which are considered valid to access and act as if the descriptor were bound to nothing.

If the **VkPhysicalDeviceRobustness2FeaturesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDeviceRobustness2FeaturesEXT** **can** also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

---

### Valid Usage

- VUID-VkPhysicalDeviceRobustness2FeaturesEXT-robustBufferAccess2-04000
  
  If **robustBufferAccess2** is enabled then **robustBufferAccess** must also be enabled

---

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceRobustness2FeaturesEXT-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_FEATURES_EXT**

The **VkPhysicalDeviceImageRobustnessFeaturesEXT** structure is defined as:

```c
// Provided by VK_EXT_image_robustness
typedef struct VkPhysicalDeviceImageRobustnessFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           robustImageAccess;
} VkPhysicalDeviceImageRobustnessFeaturesEXT;
```

This structure describes the following feature:

- **sType** is the type of this structure.

- **pNext** is **NULL** or a pointer to a structure extending this structure.

- **robustImageAccess** indicates whether image accesses are tightly bounds-checked against the
dimensions of the image view. Invalid texels resulting from out of bounds image loads will be replaced as described in Texel Replacement, with either (0,0,1) or (0,0,0) values inserted for missing G, B, or A components based on the format.

If the `VkPhysicalDeviceImageRobustnessFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceImageRobustnessFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImageRobustnessFeaturesEXT-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES_EXT`

The `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_shader_terminate_invocation
typedef struct VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           shaderTerminateInvocation;
} VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderTerminateInvocation` specifies whether the implementation supports SPIR-V modules that use the `SPV_KHR_terminate_invocation` extension.

If the `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION_FEATURES_KHR`

The `VkPhysicalDeviceCustomBorderColorFeaturesEXT` structure is defined as:
This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **customBorderColors** indicates that the implementation supports providing a `borderColor` value with one of the following values at sampler creation time:
  - `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT`
  - `VK_BORDER_COLOR_INT_CUSTOM_EXT`
- **customBorderColorWithoutFormat** indicates that explicit formats are not required for custom border colors and the value of the `format` member of the `VkSamplerCustomBorderColorCreateInfoEXT` structure may be `VK_FORMAT_UNDEFINED`. If this feature bit is not set, applications must provide the `VkFormat` of the image view(s) being sampled by this sampler in the `format` member of the `VkSamplerCustomBorderColorCreateInfoEXT` structure.

If the `VkPhysicalDeviceCustomBorderColorFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceCustomBorderColorFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceCustomBorderColorFeaturesEXT-sType-sType**

  *sType must be* `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_FEATURES_EXT`

The `VkPhysicalDevicePortabilitySubsetFeaturesKHR` structure is defined as:
typedef struct VkPhysicalDevicePortabilitySubsetFeaturesKHR {
    VkStructureType     sType;
    void*               pNext;
    VkBool32            constantAlphaColorBlendFactors;
    VkBool32            events;
    VkBool32            imageViewFormatReinterpretation;
    VkBool32            imageViewFormatSwizzle;
    VkBool32            imageView2DOn3DImage;
    VkBool32            multisampleArrayImage;
    VkBool32            mutableComparisonSamplers;
    VkBool32            pointPolygons;
    VkBool32            samplerMipLodBias;
    VkBool32            separateStencilMaskRef;
    VkBool32            shaderSampleRateInterpolationFunctions;
    VkBool32            tessellationIsolines;
    VkBool32            tessellationPointMode;
    VkBool32            triangleFans;
    VkBool32            vertexAttributeAccessBeyondStride;
} VkPhysicalDevicePortabilitySubsetFeaturesKHR;

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **constantAlphaColorBlendFactors** indicates whether this implementation supports constant *alpha Blend Factors* used as source or destination *color Blending*.
- **events** indicates whether this implementation supports synchronization using *Events*.
- **imageViewFormatReinterpretation** indicates whether this implementation supports a *VkImageView* being created with a texel format containing a different number of components, or a different number of bits in each component, than the texel format of the underlying *VkImage*.
- **imageViewFormatSwizzle** indicates whether this implementation supports remapping format components using *VkImageViewCreateInfo::components*.
- **imageView2DOn3DImage** indicates whether this implementation supports a *VkImage* being created with the *VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT* flag set, permitting a 2D or 2D array image view to be created on a 3D *VkImage*.
- **multisampleArrayImage** indicates whether this implementation supports a *VkImage* being created as a 2D array with multiple samples per texel.
- **mutableComparisonSamplers** indicates whether this implementation allows descriptors with comparison samplers to be updated.
- **pointPolygons** indicates whether this implementation supports *Rasterization* using a point Polygon Mode.
- **samplerMipLodBias** indicates whether this implementation supports setting a mipmap LOD bias value when creating a sampler.
• `separateStencilMaskRef` indicates whether this implementation supports separate front and back Stencil Test reference values.

• `shaderSampleRateInterpolationFunctions` indicates whether this implementation supports fragment shaders which use the InterpolationFunction capability and the extended instructions `InterpolateAtCentroid`, `InterpolateAtOffset`, and `InterpolateAtSample` from the GLSL.std.450 extended instruction set. This member is only meaningful if the `sampleRateShading` feature is supported.

• `tessellationIsolines` indicates whether this implementation supports isoline output from the Tessellation stage of a graphics pipeline. This member is only meaningful if `tessellation shaders` are supported.

• `tessellationPointMode` indicates whether this implementation supports point output from the Tessellation stage of a graphics pipeline. This member is only meaningful if `tessellation shaders` are supported.

• `triangleFans` indicates whether this implementation supports Triangle Fans primitive topology.

• `vertexAttributeAccessBeyondStride` indicates whether this implementation supports accessing a vertex input attribute beyond the stride of the corresponding vertex input binding.

If the `VkPhysicalDevicePortabilitySubsetFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDevicePortabilitySubsetFeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDevicePortabilitySubsetFeaturesKHR-sType-sType

  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PORTABILITY_SUBSET_FEATURES_KHR`

The `VkPhysicalDevicePerformanceQueryFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkPhysicalDevicePerformanceQueryFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 performanceCounterQueryPools;
    VkBool32 performanceCounterMultipleQueryPools;
} VkPhysicalDevicePerformanceQueryFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `performanceCounterQueryPools` indicates whether the implementation supports performance counter query pools.
• **performanceCounterMultipleQueryPools** indicates whether the implementation supports using multiple performance query pools in a primary command buffer and secondary command buffers executed within it.

If the **VkPhysicalDevicePerformanceQueryFeaturesKHR** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDevicePerformanceQueryFeaturesKHR** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDevicePerformanceQueryFeaturesKHR-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR**

The **VkPhysicalDevice4444FormatsFeaturesEXT** structure is defined as:

```c
// Provided by VK_EXT_4444_formats
typedef struct VkPhysicalDevice4444FormatsFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           formatA4R4G4B4;
    VkBool32           formatA4B4G4R4;
} VkPhysicalDevice4444FormatsFeaturesEXT;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **formatA4R4G4B4** indicates that the implementation **must** support using a **VkFormat** of **VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT** with at least the following **VkFormatFeatureFlagBits**:
  - **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT**
  - **VK_FORMAT_FEATURE_BLIT_SRC_BIT**
  - **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**
- **formatA4B4G4R4** indicates that the implementation **must** support using a **VkFormat** of **VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT** with at least the following **VkFormatFeatureFlagBits**:
  - **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT**
  - **VK_FORMAT_FEATURE_BLIT_SRC_BIT**
  - **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT**

If the **VkPhysicalDevice4444FormatsFeaturesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceFeatures2** structure passed to **vkGetPhysicalDeviceFeatures2**, it is filled in to indicate whether each corresponding feature is supported. **VkPhysicalDevice4444FormatsFeaturesEXT** can also be used in the **pNext** chain of **VkDeviceCreateInfo** to selectively enable these features.
Valid Usage (Implicit)

- VUID-VkPhysicalDevice4444FormatsFeaturesEXT-sType-sType

\textbf{sType must} be \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_4444_FORMATS_FEATURES_EXT}

The \texttt{VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE} structure is defined as:

\begin{verbatim}
// Provided by VK_VALVE_mutable_descriptor_type
typedef struct VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           mutableDescriptorType;
} VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE;
\end{verbatim}

This structure describes the following feature:

- \textbf{sType} is the type of this structure.
- \textbf{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \textbf{mutableDescriptorType} indicates that the implementation \textbf{must} support using the \texttt{VkDescriptorType} of \texttt{VK_DESCRIPTOR_TYPE_MUTABLE_VALVE} with at least the following descriptor types, where any combination of the types \textbf{must} be supported:
  - \texttt{VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE}
  - \texttt{VK_DESCRIPTOR_TYPE_STORAGE_IMAGE}
  - \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER}
  - \texttt{VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER}
  - \texttt{VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER}
  - \texttt{VK_DESCRIPTOR_TYPE_STORAGE_BUFFER}
- Additionally, \textbf{mutableDescriptorType} indicates that:
  - Non-uniform descriptor indexing \textbf{must} be supported if all descriptor types in a \texttt{VkMutableDescriptorTypeListVALVE} for \texttt{VK_DESCRIPTOR_TYPE_MUTABLE_VALVE} have the corresponding non-uniform indexing features enabled in \texttt{VkPhysicalDeviceDescriptorIndexingFeatures}.
  - \texttt{VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT} with \textbf{descriptorType} of \texttt{VK_DESCRIPTOR_TYPE_MUTABLE_VALVE} relaxes the list of required descriptor types to the descriptor types which have the corresponding update-after-bind feature enabled in \texttt{VkPhysicalDeviceDescriptorIndexingFeatures}.
  - Dynamically uniform descriptor indexing \textbf{must} be supported if all descriptor types in a \texttt{VkMutableDescriptorTypeListVALVE} for \texttt{VK_DESCRIPTOR_TYPE_MUTABLE_VALVE} have the corresponding dynamic indexing features enabled.
  - \texttt{VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE} \textbf{must} be supported.
  - \texttt{VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE} \textbf{must} be supported.
If the `VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE-sType-sType`
  
  The `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MUTABLE_DESCRIPTOR_TYPE_FEATURES_VALVE`.

The `VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_workgroup_memory_explicit_layout
typedef struct VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 workgroupMemoryExplicitLayout;
    VkBool32 workgroupMemoryExplicitLayoutScalarBlockLayout;
    VkBool32 workgroupMemoryExplicitLayout8BitAccess;
    VkBool32 workgroupMemoryExplicitLayout16BitAccess;
} VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR;
```

This structure describes the following features:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `workgroupMemoryExplicitLayout` indicates whether the implementation supports the SPIR-V `WorkgroupMemoryExplicitLayoutKHR` capability.
- `workgroupMemoryExplicitLayoutScalarBlockLayout` indicates whether the implementation supports scalar alignment for laying out Workgroup Blocks.
- `workgroupMemoryExplicitLayout8BitAccess` indicates whether objects in the `Workgroup` storage class with the `Block` decoration can have 8-bit integer members. If this feature is not enabled, 8-bit integer members must not be used in such objects. This also indicates whether shader modules can declare the `WorkgroupMemoryExplicitLayout8BitAccessKHR` capability.
- `workgroupMemoryExplicitLayout16BitAccess` indicates whether objects in the `Workgroup` storage class with the `Block` decoration can have 16-bit integer and 16-bit floating-point members. If this feature is not enabled, 16-bit integer or 16-bit floating-point members must not be used in such objects. This also indicates whether shader modules can declare the `WorkgroupMemoryExplicitLayout16BitAccessKHR` capability.

If the `VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported.
VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR-sType-sType
  sType must be
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_WORKGROUP_MEMORY_EXPLICIT_LAYOUT_FEATURES_KHR`

The `VkPhysicalDeviceSynchronization2FeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_synchronization2
typedef struct VkPhysicalDeviceSynchronization2FeaturesKHR {
    VkStructureType sType;
    void*           pNext;
    VkBool32       synchronization2;
} VkPhysicalDeviceSynchronization2FeaturesKHR;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `synchronization2` indicates whether the implementation supports the new set of synchronization commands introduced in VK_KHR_synchronization2.

If the `VkPhysicalDeviceSynchronization2FeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceSynchronization2FeaturesKHR` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceSynchronization2FeaturesKHR-sType-sType
  sType must be
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SYNCHRONIZATION_2_FEATURES_KHR`

The `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_vertex_input_dynamic_state
typedef struct VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT {
    VkStructureType sType;
    void*           pNext;
    VkBool32       vertexInputDynamicState;
} VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT;
```
This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **vertexInputDynamicState** indicates that the implementation supports the following dynamic states:
  - **VK_DYNAMIC_STATE_VERTEX_INPUT_EXT**

If the `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_INPUT_DYNAMIC_STATE_FEATURES_EXT`

The `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` structure is defined as:

```c
// Provided by VK_KHR_fragment_shading_rate
typedef struct VkPhysicalDeviceFragmentShadingRateFeaturesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 pipelineFragmentShadingRate;
    VkBool32 primitiveFragmentShadingRate;
    VkBool32 attachmentFragmentShadingRate;
} VkPhysicalDeviceFragmentShadingRateFeaturesKHR;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **pipelineFragmentShadingRate** indicates that the implementation supports the **pipeline fragment shading rate**.
- **primitiveFragmentShadingRate** indicates that the implementation supports the **primitive fragment shading rate**.
- **attachmentFragmentShadingRate** indicates that the implementation supports the **attachment fragment shading rate**.

If the `VkPhysicalDeviceFragmentShadingRateFeaturesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported.
VkPhysicalDeviceFragmentShadingRateFeaturesKHR can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceFragmentShadingRateFeaturesKHR-sType-sType**
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_FEATURES_KHR

The VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV structure is defined as:

```c
// Provided by VK_NV_fragment_shading_rate_enums
typedef struct VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           fragmentShadingRateEnums;
    VkBool32           supersampleFragmentShadingRates;
    VkBool32           noInvocationFragmentShadingRates;
} VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV;
```

This structure describes the following features:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **fragmentShadingRateEnums** indicates that the implementation supports specifying fragment shading rates using the VkFragmentShadingRateNV enumerated type.
- **supersampleFragmentShadingRates** indicates that the implementation supports fragment shading rate enum values indicating more than one invocation per fragment.
- **noInvocationFragmentShadingRates** indicates that the implementation supports a fragment shading rate enum value indicating that no fragment shaders should be invoked when that shading rate is used.

If the VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceFragmentShadingRateEnumsFeaturesNV-sType-sType**
  - sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_ENUMS_FEATURES_NV

The VkPhysicalDeviceInheritedViewportScissorFeaturesNV structure is defined as:
typedef struct VkPhysicalDeviceInheritedViewportScissorFeaturesNV {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           inheritedViewportScissor2D;
} VkPhysicalDeviceInheritedViewportScissorFeaturesNV;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **inheritedViewportScissor2D** indicates whether secondary command buffers can inherit most of the dynamic state affected by VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT, VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT, VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT, VK_DYNAMIC_STATE_VIEWPORT or VK_DYNAMIC_STATE_SCISSOR, from a primary command buffer.

If the VkPhysicalDeviceInheritedViewportScissorFeaturesNV structure is included in the pNext chain of the VkPhysicalDeviceFeatures2 structure passed to vkGetPhysicalDeviceFeatures2, it is filled in to indicate whether each corresponding feature is supported. VkPhysicalDeviceInheritedViewportScissorFeaturesNV can also be used in the pNext chain of VkDeviceCreateInfo to selectively enable these features.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceInheritedViewportScissorFeaturesNV-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INHERITED_VIEWPORT_SCISSOR_FEATURES_NV

The VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT structure is defined as:

typedef struct VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           ycbcr2plane444Formats;
} VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **ycbcr2plane444Formats** indicates that the implementation supports the following 2-plane 444 Y′C_{B}C_{R} formats:
  - VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT
VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT

If the `VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_2_PLANE_444_FORMATS_FEATURES_EXT`

The `VkPhysicalDeviceColorWriteEnableFeaturesEXT` structure is defined as:

```c
// Provided by VK_EXT_color_write_enable
typedef struct VkPhysicalDeviceColorWriteEnableFeaturesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           colorWriteEnable;
} VkPhysicalDeviceColorWriteEnableFeaturesEXT;
```

This structure describes the following feature:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `colorWriteEnable` indicates that the implementation supports the dynamic state `VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT`.

If the `VkPhysicalDeviceColorWriteEnableFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceColorWriteEnableFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceColorWriteEnableFeaturesEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COLOR_WRITE_ENABLE_FEATURES_EXT`

The `VkPhysicalDeviceProvokingVertexFeaturesEXT` structure is defined as:
typedef struct VkPhysicalDeviceProvokingVertexFeaturesEXT {
  VkStructureType sType;
  void* pNext;
  VkBool32 provokingVertexLast;
  VkBool32 transformFeedbackPreservesProvokingVertex;
} VkPhysicalDeviceProvokingVertexFeaturesEXT;

This structure describes the following feature:

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **provokingVertexLast** indicates whether the implementation supports the `VK_PROVOKING_VERTEX_MODE_LAST_VERTEX_EXT` provoking vertex mode for flat shading.
- **transformFeedbackPreservesProvokingVertex** indicates that the order of vertices within each primitive written by transform feedback will preserve the provoking vertex. This does not apply to triangle fan primitives when `transformFeedbackPreservesTriangleFanProvokingVertex` is `VK_FALSE`. `transformFeedbackPreservesProvokingVertex` must be `VK_FALSE` when the VK_EXT_transform_feedback extension is not supported.

If the `VkPhysicalDeviceProvokingVertexFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceFeatures2` structure passed to `vkGetPhysicalDeviceFeatures2`, it is filled in to indicate whether each corresponding feature is supported. `VkPhysicalDeviceProvokingVertexFeaturesEXT` can also be used in the `pNext` chain of `VkDeviceCreateInfo` to selectively enable these features.

When `VkPhysicalDeviceProvokingVertexFeaturesEXT` is in the `pNext` chain of `VkDeviceCreateInfo` but the transform feedback feature is not enabled, the value of `transformFeedbackPreservesProvokingVertex` is ignored.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceProvokingVertexFeaturesEXT-sType-sType
  
  *sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROVOKING_VERTEX_FEATURES_EXT*

### 41.1. Feature Requirements

All Vulkan graphics implementations **must** support the following features:

- **robustBufferAccess**, unless the `VK_KHR_portability_subset` extension is enabled.
- **multiview**, if Vulkan 1.1 is supported.
- **shaderDrawParameters**, if the `VK_KHR_shader_draw_parameters` extension is supported.
- **uniformBufferStandardLayout**, if Vulkan 1.2 or the `VK_KHR_uniform_buffer_standard_layout` extension is supported.
- **variablePointersStorageBuffer**, if the `VK_KHR_variable_pointers` extension is supported.
- **storageBuffer8BitAccess**, if the `VK_KHR_8bit_storage` extension is supported.
- **StorageBuffer8BitAccess**, if `uniformAndStorageBuffer8BitAccess` is enabled.
- If the `VK_EXT_descriptor_indexing` extension is supported:
  - `shaderSampledImageArrayDynamicIndexing`
  - `shaderStorageBufferArrayDynamicIndexing`
  - `shaderUniformTexelBufferArrayDynamicIndexing`
  - `shaderStorageTexelBufferArrayDynamicIndexing`
  - `shaderSampledImageArrayNonUniformIndexing`
  - `shaderStorageBufferArrayNonUniformIndexing`
  - `shaderUniformTexelBufferArrayNonUniformIndexing`
  - `descriptorBindingSampledImageUpdateAfterBind`
  - `descriptorBindingStorageImageUpdateAfterBind`
  - `descriptorBindingStorageBufferUpdateAfterBind` (see also `robustBufferAccessUpdateAfterBind`)
  - `descriptorBindingUniformTexelBufferUpdateAfterBind` (see also `robustBufferAccessUpdateAfterBind`)
  - `descriptorBindingStorageTexelBufferUpdateAfterBind` (see also `robustBufferAccessUpdateAfterBind`)
  - `descriptorBindingUpdateUnusedWhilePending`
  - `descriptorBindingPartiallyBound`
  - `runtimeDescriptorArray`
- **inlineUniformBlock**, if the `VK_EXT_inline_uniform_block` extension is supported.
- **descriptorBindingInlineUniformBlockUpdateAfterBind**, if the `VK_EXT_inline_uniform_block` extension is supported; and if the `VK_EXT_descriptor_indexing` extension is supported.
- **scalarBlockLayout**, if the `VK_EXT_scalar_block_layout` extension is supported.
- **subgroupSizeControl**, if the `VK_EXT_subgroup_size_control` extension is supported.
- **computeFullSubgroups**, if the `VK_EXT_subgroup_size_control` extension is supported.
- **deviceMemoryReport**, if the `VK_EXT_device_memory_report` extension is supported.
- **imagelessFramebuffer**, if Vulkan 1.2 or the `VK_KHR_imageless_framebuffer` extension is supported.
- **separateDepthStencilLayouts**, if Vulkan 1.2 or the `VK_KHR_separate_depthStencil_layouts` extension is supported.
- **hostQueryReset**, if Vulkan 1.2 or the `VK_EXT_host_query_reset` extension is supported.
- **timelineSemaphore**, if Vulkan 1.2 or the `VK_KHR_timeline_semaphore` extension is supported.
- If the `VK_KHR_acceleration_structure` extension is supported:
  - `accelerationStructure`
  - All the features required by the `VK_EXT_descriptor_indexing` extension.
• descriptorBindingAccelerationStructureUpdateAfterBind
• bufferDeviceAddress from the VK_KHR_buffer_device_address extension.

• If the VK_KHR_ray_tracing_pipeline extension is supported:
  • accelerationStructure from VK_KHR_acceleration_structure
  • rayTracingPipeline
  • rayTracingPipelineTraceRaysIndirect
  • rayTraversalPrimitiveCulling, if rayQuery is supported from VK_KHR_ray_query
  • the VK_KHR_pipeline_library extension.

• If the VK_KHR_ray_query extension is supported:
  • accelerationStructure from VK_KHR_acceleration_structure
  • rayQuery

• pipelineCreationCacheControl, if the VK_EXT_pipeline_creation_cache_control extension is supported.

• shaderSubgroupExtendedTypes, if Vulkan 1.2 or the VK_KHR_shader_subgroup_extended_types extension is supported.

• samplerYcbcrConversion, if the VK_KHR_sampler_ycbcr_conversion extension is supported.

• pipelineExecutableInfo, if the VK_KHR_pipeline_executable_properties extension is supported.

• textureCompressionASTC_HDR, if the VK_EXT_texture_compression_astc_hdr extension is supported.

• depthClipEnable, if the VK_EXT_depth_clip_enable extension is supported.

• memoryPriority, if the VK_EXT_memory_priority extension is supported.

• ycbcrImageArrays, if the VK_EXT_ycbcr_image_arrays extension is supported.

• indexTypeUint8, if the VK_EXT_index_type_uint8 extension is supported.

• shaderDemoteToHelperInvocation, if the VK_EXT_shader_demote_to_helper_invocation extension is supported.

• texelBufferAlignment, if the VK_EXT_texel_buffer_alignment extension is supported.

• vulkanMemoryModel, if the VK_KHR_vulkan_memory_model extension is supported.

• bufferDeviceAddress, if the VK_KHR_buffer_device_address extension is supported.

• performanceCounterQueryPools, if the VK_KHR_performance_query extension is supported.

• transformFeedback, if the VK_EXT_transform_feedback extension is supported.

• conditionalRendering, if the VK_EXT_conditional_rendering extension is supported.

• vertexAttributeInstanceRateDivisor, if the VK_EXT_vertex_attribute_divisor extension is supported.

• fragmentDensityMap, if the VK_EXT_fragment_density_map extension is supported.

• shaderSubgroupClock, if the VK_KHR_shader_clock extension is supported.

• shaderBufferInt64Atomics, if the VK_KHR_shader_atomic_int64 extension is supported.

• shaderFloat16 or shaderInt8, if the VK_KHR_shader_float16_int8 extension is supported.
• `fragmentShaderSampleInterlock` or `fragmentShaderPixelInterlock` or `fragmentShaderShadingRateInterlock`, if the `VK_EXT_fragment_shader_interlock` extension is supported.

• `rectangularLines` or `bresenhamLines` or `smoothLines` or `stippledRectangularLines` or `stippledBresenhamLines` or `stippledSmoothLines`, if the `VK_EXT_line_rasterization` extension is supported.

• `storageBuffer16BitAccess`, if the `VK_KHR_16bit_storage` extension is supported.

• `storageBuffer16BitAccess`, if `uniformAndStorageBuffer16BitAccess` is enabled.

• `robustImageAccess`, if the `VK_EXT_image_robustness` extension is supported.

• `formatA4R4G4B4`, if the `VK_EXT_4444_formats` extension is supported.

• `mutableDescriptorType`, if the `VK_VALVE_mutable_descriptor_type` extension is supported.

• `shaderInt64` and `shaderImageInt64Atomics`, if the `VK_EXT_shader_image_atomic_int64` extension is supported.

• `pipelineFragmentShadingRate`, if the `VK_KHR_fragment_shading_rate` extension is supported.

• `shaderTerminateInvocation` if the `VK_KHR_shader_terminate_invocation` extension is supported.

• `shaderZeroInitializeWorkgroupMem`, if the `VK_KHR_zero_initialize_workgroup_memory` extension is supported.

• `workgroupMemoryExplicitLayout`, if the `VK_KHR_workgroup_memory_explicit_layout` extension is supported.

• `vertexInputDynamicState`, if the `VK_EXT_vertex_input_dynamic_state` extension is supported.

• `synchronization2` if the `VK_KHR_synchronization2` extension is supported.

• `provokingVertexLast`, if the `VK_EXT_provoking_vertex` extension is supported.

All other features defined in the Specification are **optional**.

If the `VkPhysicalDeviceProvokingVertexFeaturesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.
Chapter 42. Limits

*Limits* are implementation-dependent minimums, maximums, and other device characteristics that an application *may* need to be aware of.

**Note**

Limits are reported via the basic `VkPhysicalDeviceLimits` structure as well as the extensible structure `VkPhysicalDeviceProperties2`, which was added in `VK_KHR_get_physical_device_properties2` and included in Vulkan 1.1. When limits are added in future Vulkan versions or extensions, each extension *should* introduce one new limit structure, if needed. This structure *can* be added to the `pNext` chain of the `VkPhysicalDeviceProperties2` structure.

The `VkPhysicalDeviceLimits` structure is defined as:

```
// Provided by VK_VERSION_1_0
typedef struct VkPhysicalDeviceLimits {
    uint32_t maxImageDimension1D;
    uint32_t maxImageDimension2D;
    uint32_t maxImageDimension3D;
    uint32_t maxImageDimensionCube;
    uint32_t maxImageArrayLayers;
    uint32_t maxTexelBufferElements;
    uint32_t maxUniformBufferRange;
    uint32_t maxStorageBufferRange;
    uint32_t maxPushConstantsSize;
    uint32_t maxMemoryAllocationCount;
    uint32_t maxSamplerAllocationCount;
    VkDeviceSize bufferImageGranularity;
    VkDeviceSize sparseAddressSpaceSize;
    uint32_t maxBoundDescriptorSets;
    uint32_t maxPerStageDescriptorSamplers;
    uint32_t maxPerStageDescriptorUniformBuffers;
    uint32_t maxPerStageDescriptorStorageBuffers;
    uint32_t maxPerStageDescriptorSampledImages;
    uint32_t maxPerStageDescriptorStorageImages;
    uint32_t maxPerStageDescriptorInputAttachments;
    uint32_t maxVertexInputAttributes;
    uint32_t maxVertexInputBindings;
    uint32_t maxVertexInputAttributeOffset;
} VkPhysicalDeviceLimits;
```
uint32_t maxVertexInputBindingStride;
uint32_t maxVertexOutputComponents;
uint32_t maxTessellationGenerationLevel;
uint32_t maxTessellationPatchSize;
uint32_t maxTessellationControlPerVertexInputComponents;
uint32_t maxTessellationControlPerVertexOutputComponents;
uint32_t maxTessellationControlTotalOutputComponents;
uint32_t maxTessellationEvaluationInputComponents;
uint32_t maxTessellationEvaluationOutputComponents;
uint32_t maxGeometryShaderInvocations;
uint32_t maxGeometryInputComponents;
uint32_t maxGeometryOutputComponents;
uint32_t maxGeometryOutputVertices;
uint32_t maxGeometryTotalOutputComponents;
uint32_t maxFragmentInputComponents;
uint32_t maxFragmentOutputAttachments;
uint32_t maxFragmentDualSrcAttachments;
uint32_t maxFragmentCombinedOutputResources;
uint32_t maxComputeSharedMemorySize;
uint32_t maxComputeWorkGroupCount[3];
uint32_t maxComputeWorkGroupInvocations;
uint32_t maxComputeWorkGroupSize[3];
uint32_t subPixelPrecisionBits;
uint32_t subTexelPrecisionBits;
uint32_t mipmapPrecisionBits;
uint32_t maxDrawIndexedIndexValue;
uint32_t maxDrawIndirectCount;
float maxSamplerLodBias;
float maxSamplerAnisotropy;
uint32_t maxViewports;
uint32_t maxViewportDimensions[2];
float viewportBoundsRange[2];
uint32_t viewportSubPixelBits;
size_t minMemoryMapAlignment;
VkDeviceSize minTexelBufferOffsetAlignment;
VkDeviceSize minUniformBufferOffsetAlignment;
VkDeviceSize minStorageBufferOffsetAlignment;
int32_t minTexelOffset;
uint32_t maxTexelOffset;
int32_t minTexelGatherOffset;
uint32_t maxTexelGatherOffset;
float minInterpolationOffset;
float maxInterpolationOffset;
uint32_t subPixelInterpolationOffsetBits;
uint32_t maxFramebufferWidth;
uint32_t maxFramebufferHeight;
uint32_t maxFramebufferLayers;
VkSampleCountFlags framebufferColorSampleCounts;
VkSampleCountFlags framebufferDepthSampleCounts;
VkSampleCountFlags framebufferStencilSampleCounts;
The `VkPhysicalDeviceLimits` are properties of the physical device. These are available in the `limits` member of the `VkPhysicalDeviceProperties` structure which is returned from `vkGetPhysicalDeviceProperties`.

- `maxImageDimension1D` is the largest dimension (width) that is guaranteed to be supported for all images created with an `imageType` of `VK_IMAGE_TYPE_1D`. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using `vkGetPhysicalDeviceImageFormatProperties`.

- `maxImageDimension2D` is the largest dimension (width or height) that is guaranteed to be supported for all images created with an `imageType` of `VK_IMAGE_TYPE_2D` and without `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` set in `flags`. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using `vkGetPhysicalDeviceImageFormatProperties`.

- `maxImageDimension3D` is the largest dimension (width, height, or depth) that is guaranteed to be supported for all images created with an `imageType` of `VK_IMAGE_TYPE_3D`. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using `vkGetPhysicalDeviceImageFormatProperties`.

- `maxImageDimensionCube` is the largest dimension (width or height) that is guaranteed to be supported for all images created with an `imageType` of `VK_IMAGE_TYPE_2D` and with `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT` set in `flags`. Some combinations of image parameters (format, usage, etc.) may allow support for larger dimensions, which can be queried using `vkGetPhysicalDeviceImageFormatProperties`.

- `maxImageArrayLayers` is the maximum number of layers (`arrayLayers`) for an image.
• **maxTexelBufferElements** is the maximum number of addressable texels for a buffer view created on a buffer which was created with the `VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT` or `VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT` set in the `usage` member of the `VkBufferCreateInfo` structure.

• **maxUniformBufferRange** is the maximum value that **can** be specified in the `range` member of a `VkDescriptorBufferInfo` structure passed to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`.

• **maxStorageBufferRange** is the maximum value that **can** be specified in the `range` member of a `VkDescriptorBufferInfo` structure passed to `vkUpdateDescriptorSets` for descriptors of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`.

• **maxPushConstantsSize** is the maximum size, in bytes, of the pool of push constant memory. For each of the push constant ranges indicated by the `pPushConstantRanges` member of the `VkPipelineLayoutCreateInfo` structure, `(offset + size)` **must** be less than or equal to this limit.

• **maxMemoryAllocationCount** is the maximum number of device memory allocations, as created by `vkAllocateMemory`, which **can** simultaneously exist.

• **maxSamplerAllocationCount** is the maximum number of sampler objects, as created by `vkCreateSampler`, which **can** simultaneously exist on a device.

• **bufferImageGranularity** is the granularity, in bytes, at which buffer or linear image resources, and optimal image resources **can** be bound to adjacent offsets in the same `VkDeviceMemory` object without aliasing. See **Buffer-Image Granularity** for more details.

• **sparseAddressSpaceSize** is the total amount of address space available, in bytes, for sparse memory resources. This is an upper bound on the sum of the sizes of all sparse resources, regardless of whether any memory is bound to them.

• **maxBoundDescriptorSets** is the maximum number of descriptor sets that **can** be simultaneously used by a pipeline. All `DescriptorSet` decorations in shader modules **must** have a value less than `maxBoundDescriptorSets`. See **Descriptor Sets**.

• **maxPerStageDescriptorSamplers** is the maximum number of samplers that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_SAMPLER` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See **Sampler** and **Combined Image Sampler**.

• **maxPerStageDescriptorUniformBuffers** is the maximum number of uniform buffers that **can** be accessible to a single shader stage in a pipeline layout. Descriptors with a type of `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` count against this limit. Only descriptors in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit. A descriptor is accessible to a shader stage when the `stageFlags` member of the `VkDescriptorSetLayoutBinding` structure has the bit for that shader stage set. See **Uniform Buffer** and **Dynamic Uniform Buffer**.

• **maxPerStageDescriptorStorageBuffers** is the maximum number of storage buffers that **can** be
accessible to a single shader stage in a pipeline layout. Descriptors with a type of 
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against 
this limit. Only descriptors in descriptor set layouts created without the 
VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. A 
descriptor is accessible to a pipeline shader stage when the stageFlags member of the 
VkDescriptorsetLayoutBinding structure has the bit for that shader stage set. See Storage Buffer 
and Dynamic Storage Buffer.

- maxPerStageDescriptorSampledImages is the maximum number of sampled images that can be 
accessible to a single shader stage in a pipeline layout. Descriptors with a type of 
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or 
VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER count against this limit. Only descriptors in descriptor 
set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit 
set count against this limit. A descriptor is accessible to a pipeline shader stage when the stageFlags member of the 
VkDescriptorsetLayoutBinding structure has the bit for that shader stage set. See Combined Image Sampler, Sampled Image, and Uniform Texel Buffer.

- maxPerStageDescriptorStorageImages is the maximum number of storage images that can be 
accessible to a single shader stage in a pipeline layout. Descriptors with a type of 
VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER count against 
this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the stageFlags member of the 
VkDescriptorsetLayoutBinding structure has the bit for that shader stage set. See Storage Image, and Storage Texel Buffer.

- maxPerStageDescriptorInputAttachments is the maximum number of input attachments that can be 
accessible to a single shader stage in a pipeline layout. Descriptors with a type of 
VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. Only descriptors in descriptor set 
layers created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. A descriptor is accessible to a pipeline shader stage when the stageFlags member of the 
VkDescriptorsetLayoutBinding structure has the bit for that shader stage set. These are only supported for the fragment stage. See Input Attachment.

- maxPerStageResources is the maximum number of resources that can be accessible to a single 
shader stage in a pipeline layout. Descriptors with a type of 
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 
VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 
VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. Only descriptors in descriptor set layouts created without the 
VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. For 
the fragment shader stage the framebuffer color attachments also count against this limit.

- maxDescriptorSetSamplers is the maximum number of samplers that can be included in a 
pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_SAMPLER or 
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER count against this limit. Only descriptors in 
descr...

---

Chapter 42. Limits | 2189
• **maxDescriptorSetUniformBuffers** is the maximum number of uniform buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Uniform Buffer and Dynamic Uniform Buffer.

• **maxDescriptorSetUniformBuffersDynamic** is the maximum number of dynamic uniform buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Dynamic Uniform Buffer.

• **maxDescriptorSetStorageBuffers** is the maximum number of storage buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Storage Buffer and Dynamic Storage Buffer.

• **maxDescriptorSetStorageBuffersDynamic** is the maximum number of dynamic storage buffers that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Dynamic Storage Buffer.

• **maxDescriptorSetSampledImages** is the maximum number of sampled images that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Combined Image Sampler, Sampled Image, and Uniform Texel Buffer.

• **maxDescriptorSetStorageImages** is the maximum number of storage images that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Storage Image, and Storage Texel Buffer.

• **maxDescriptorSetInputAttachments** is the maximum number of input attachments that can be included in a pipeline layout. Descriptors with a type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. Only descriptors in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit. See Input Attachment.

• **maxVertexInputAttributes** is the maximum number of vertex input attributes that can be specified for a graphics pipeline. These are described in the array of VkVertexInputAttributeDescription structures that are provided at graphics pipeline creation time via the pVertexAttributeDescriptions member of the VkPipelineVertexInputStateCreateInfo structure. See Vertex Attributes and Vertex Input Description.
• **maxVertexInputBindings** is the maximum number of vertex buffers that can be specified for providing vertex attributes to a graphics pipeline. These are described in the array of `VkVertexInputBindingDescription` structures that are provided at graphics pipeline creation time via the `pVertexBindingDescriptions` member of the `VkPipelineVertexInputStateCreateInfo` structure. The `binding` member of `VkVertexInputBindingDescription` must be less than or equal to this limit. See **Vertex Input Description**.

• **maxVertexInputAttributeOffset** is the maximum vertex input attribute offset that can be added to the vertex input binding stride. The `offset` member of the `VkVertexInputAttributeDescription` structure must be less than or equal to this limit. See **Vertex Input Description**.

• **maxVertexInputBindingStride** is the maximum vertex input binding stride that can be specified in a vertex input binding. The `stride` member of the `VkVertexInputBindingDescription` structure must be less than or equal to this limit. See **Vertex Input Description**.

• **maxVertexOutputComponents** is the maximum number of components of output variables which can be output by a vertex shader. See **Vertex Shaders**.

• **maxTessellationGenerationLevel** is the maximum tessellation generation level supported by the fixed-function tessellation primitive generator. See **Tessellation**.

• **maxTessellationPatchSize** is the maximum patch size, in vertices, of patches that can be processed by the tessellation control shader and tessellation primitive generator. The `patchControlPoints` member of the `VkPipelineTessellationStateCreateInfo` structure specified at pipeline creation time and the value provided in the `OutputVertices` execution mode of shader modules must be less than or equal to this limit. See **Tessellation**.

• **maxTessellationControlPerVertexInputComponents** is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation control shader stage.

• **maxTessellationControlPerVertexOutputComponents** is the maximum number of components of per-vertex output variables which can be output from the tessellation control shader stage.

• **maxTessellationControlPerPatchOutputComponents** is the maximum number of components of per-patch output variables which can be output from the tessellation control shader stage.

• **maxTessellationControlTotalOutputComponents** is the maximum total number of components of per-vertex and per-patch output variables which can be output from the tessellation control shader stage.

• **maxTessellationEvaluationInputComponents** is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation evaluation shader stage.

• **maxTessellationEvaluationOutputComponents** is the maximum number of components of per-vertex output variables which can be output from the tessellation evaluation shader stage.

• **maxGeometryShaderInvocations** is the maximum invocation count supported for instanced geometry shaders. The value provided in the `Invocations` execution mode of shader modules must be less than or equal to this limit. See **Geometry Shading**.

• **maxGeometryInputComponents** is the maximum number of components of input variables which can be provided as inputs to the geometry shader stage.

• **maxGeometryOutputComponents** is the maximum number of components of output variables which
can be output from the geometry shader stage.

- **maxGeometryOutputVertices** is the maximum number of vertices which can be emitted by any geometry shader.

- **maxGeometryTotalOutputComponents** is the maximum total number of components of output variables, across all emitted vertices, which can be output from the geometry shader stage.

- **maxFragmentInputComponents** is the maximum number of components of input variables which can be provided as inputs to the fragment shader stage.

- **maxFragmentOutputAttachments** is the maximum number of output attachments which can be written to by the fragment shader stage.

- **maxFragmentDualSrcAttachments** is the maximum number of output attachments which can be written to by the fragment shader stage when blending is enabled and one of the dual source blend modes is in use. See Dual-Source Blending and dualSrcBlend.

- **maxFragmentCombinedOutputResources** is the total number of storage buffers, storage images, and output Location decorated color attachments (described in Fragment Output Interface) which can be used in the fragment shader stage.

- **maxComputeSharedMemorySize** is the maximum total storage size, in bytes, available for variables declared with the Workgroup storage class in shader modules (or with the shared storage qualifier in GLSL) in the compute shader stage. When variables declared with the Workgroup storage class are explicitly laid out (hence they are also decorated with Block), the amount of storage consumed is the size of the largest Block variable, not counting any padding at the end. The amount of storage consumed by the non-Block variables declared with the Workgroup storage class is implementation-dependent. However, the amount of storage consumed may not exceed the largest block size that would be obtained if all active non-Block variables declared with Workgroup storage class were assigned offsets in an arbitrary order by successively taking the smallest valid offset according to the Standard Storage Buffer Layout rules. (This is equivalent to using the GLSL std430 layout rules.)

- **maxComputeWorkGroupCount[3]** is the maximum number of local workgroups that can be dispatched by a single dispatch command. These three values represent the maximum number of local workgroups for the X, Y, and Z dimensions, respectively. The workgroup count parameters to the dispatch commands must be less than or equal to the corresponding limit. See Dispatching Commands.

- **maxComputeWorkGroupInvocations** is the maximum total number of compute shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the LocalSize execution mode in shader modules or by the object decorated by the WorkgroupSize decoration, must be less than or equal to this limit.

- **maxComputeWorkGroupSize[3]** is the maximum size of a local compute workgroup, per dimension. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The x, y, and z sizes, as specified by the LocalSize execution mode or by the object decorated by the WorkgroupSize decoration in shader modules, must be less than or equal to the corresponding limit.

- **subPixelPrecisionBits** is the number of bits of subpixel precision in framebuffer coordinates x_f and y_f. See Rasterization.

- **subTexelPrecisionBits** is the number of bits of precision in the division along an axis of an
image used for minification and magnification filters. \( 2^{\text{subTexelPrecisionBits}} \) is the actual number of divisions along each axis of the image represented. Sub-texel values calculated during image sampling will snap to these locations when generating the filtered results.

- \textbf{mipmapPrecisionBits} is the number of bits of division that the LOD calculation for mipmap fetching get snapped to when determining the contribution from each mipmap level to the mipmap filtered results. \( 2^{\text{mipmapPrecisionBits}} \) is the actual number of divisions.

- \textbf{maxDrawIndexedIndexValue} is the maximum index value that \textbf{can} be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of 0xFFFFFFFF. See \texttt{fullDrawIndexUint32}.

- \textbf{maxDrawIndirectCount} is the maximum draw count that is supported for indirect draw calls. See \texttt{multiDrawIndirect}.

- \textbf{maxSamplerLodBias} is the maximum absolute sampler LOD bias. The sum of the \texttt{mipLodBias} member of the \texttt{VkSamplerCreateInfo} structure and the \texttt{Bias} operand of image sampling operations in shader modules (or 0 if no \texttt{Bias} operand is provided to an image sampling operation) are clamped to the range \([-\text{maxSamplerLodBias},+\text{maxSamplerLodBias}]\). See [samplers-mipLodBias].

- \textbf{maxSamplerAnisotropy} is the maximum degree of sampler anisotropy. The maximum degree of anisotropic filtering used for an image sampling operation is the minimum of the \texttt{maxAnisotropy} member of the \texttt{VkSamplerCreateInfo} structure and this limit. See [samplers-maxAnisotropy].

- \textbf{maxViewports} is the maximum number of active viewports. The \texttt{viewportCount} member of the \texttt{VkPipelineViewportStateCreateInfo} structure that is provided at pipeline creation \textbf{must} be less than or equal to this limit.

- \textbf{maxViewportDimensions[2]} are the maximum viewport dimensions in the X (width) and Y (height) dimensions, respectively. The maximum viewport dimensions \textbf{must} be greater than or equal to the largest image which \textbf{can} be created and used as a framebuffer attachment. See Controlling the Viewport.

- \textbf{viewportBoundsRange[2]} is the [minimum, maximum] range that the corners of a viewport \textbf{must} be contained in. This range \textbf{must} be at least \([-2 \times \text{size}, 2 \times \text{size} - 1]\), where \texttt{size} = \text{max}(\text{maxViewportDimensions}[0], \text{maxViewportDimensions}[1]). See Controlling the Viewport.

\begin{itemize}
  \item \textbf{viewportSubPixelBits} is the number of bits of subpixel precision for viewport bounds. The subpixel precision that floating-point viewport bounds are interpreted at is given by this limit.
  \item \textbf{minMemoryMapAlignment} is the minimum \textbf{required} alignment, in bytes, of host visible memory allocations within the host address space. When mapping a memory allocation with \texttt{vkMapMemory}, subtracting \texttt{offset} bytes from the returned pointer will always produce an
\end{itemize}
integer multiple of this limit. See Host Access to Device Memory Objects.

- **minTexelBufferOffsetAlignment** is the minimum required alignment, in bytes, for the `offset` member of the `VkBufferViewCreateInfo` structure for texel buffers. If `texelBufferAlignment` is enabled, this limit is equivalent to the maximum of the `uniformTexelBufferOffsetAlignmentBytes` and `storageTexelBufferOffsetAlignmentBytes` members of `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT`, but smaller alignment is optionally allowed by `storageTexelBufferOffsetSingleTexelAlignment` and `uniformTexelBufferOffsetSingleTexelAlignment`. If `texelBufferAlignment` is not enabled, `VkBufferViewCreateInfo::offset` must be a multiple of this value.

- **minUniformBufferOffsetAlignment** is the minimum required alignment, in bytes, for the `offset` member of the `VkDescriptorBufferInfo` structure for uniform buffers. When a descriptor of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` or `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` is updated, the `offset` must be an integer multiple of this limit. Similarly, dynamic offsets for uniform buffers must be multiples of this limit.

- **minStorageBufferOffsetAlignment** is the minimum required alignment, in bytes, for the `offset` member of the `VkDescriptorBufferInfo` structure for storage buffers. When a descriptor of type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` is updated, the `offset` must be an integer multiple of this limit. Similarly, dynamic offsets for storage buffers must be multiples of this limit.

- **minTexelOffset** is the minimum offset value for the `ConstOffset` image operand of any of the `OpImageSample*` or `OpImageFetch*` image instructions.

- **maxTexelOffset** is the maximum offset value for the `ConstOffset` image operand of any of the `OpImageSample*` or `OpImageFetch*` image instructions.

- **minTexelGatherOffset** is the minimum offset value for the `Offset`, `ConstOffset`, or `ConstOffsets` image operands of any of the `OpImage*Gather` image instructions.

- **maxTexelGatherOffset** is the maximum offset value for the `Offset`, `ConstOffset`, or `ConstOffsets` image operands of any of the `OpImage*Gather` image instructions.

- **minInterpolationOffset** is the base minimum (inclusive) negative offset value for the `Offset` operand of the `InterpolateAtOffset` extended instruction.

- **maxInterpolationOffset** is the base maximum (inclusive) positive offset value for the `Offset` operand of the `InterpolateAtOffset` extended instruction.

- **subPixelInterpolationOffsetBits** is the number of fractional bits that the `x` and `y` offsets to the `InterpolateAtOffset` extended instruction may be rounded to as fixed-point values.

- **maxFramebufferWidth** is the maximum width for a framebuffer. The `width` member of the `VkFramebufferCreateInfo` structure must be less than or equal to this limit.

- **maxFramebufferHeight** is the maximum height for a framebuffer. The `height` member of the `VkFramebufferCreateInfo` structure must be less than or equal to this limit.

- **maxFramebufferLayers** is the maximum layer count for a layered framebuffer. The `layers` member of the `VkFramebufferCreateInfo` structure must be less than or equal to this limit.

- **framebufferColorSampleCounts** is a bitmask of `VkSampleCountFlagBits` indicating the color sample counts that are supported for all framebuffer color attachments with floating- or fixed-point formats. There is no limit that specifies the color sample counts that are supported for all
color attachments with integer formats.

- **framebufferDepthSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the supported depth sample counts for all framebuffer depth/stencil attachments, when the format includes a depth component.

- **framebufferStencilSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the supported stencil sample counts for all framebuffer depth/stencil attachments, when the format includes a stencil component.

- **framebufferNoAttachmentsSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the supported sample counts for a subpass which uses no attachments.

- **maxColorAttachments** is the maximum number of color attachments that can be used by a subpass in a render pass. The **colorAttachmentCount** member of the **VkSubpassDescription** or **VkSubpassDescription2** structure must be less than or equal to this limit.

- **sampledImageColorSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the sample counts supported for all 2D images created with **VK_IMAGE_TILING_OPTIMAL**, **usage** containing **VK_IMAGE_USAGE_SAMPLED_BIT**, and a non-integer color format.

- **sampledImageIntegerSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the sample counts supported for all 2D images created with **VK_IMAGE_TILING_OPTIMAL**, **usage** containing **VK_IMAGE_USAGE_SAMPLED_BIT**, and an integer color format.

- **sampledImageDepthSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the sample counts supported for all 2D images created with **VK_IMAGE_TILING_OPTIMAL**, **usage** containing **VK_IMAGE_USAGE_SAMPLED_BIT**, and a depth format.

- **sampledImageStencilSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the sample counts supported for all 2D images created with **VK_IMAGE_TILING_OPTIMAL**, **usage** containing **VK_IMAGE_USAGE_SAMPLED_BIT**, and a stencil format.

- **storageImageSampleCounts** is a bitmask\(^1\) of **VkSampleCountFlagBits** indicating the sample counts supported for all 2D images created with **VK_IMAGE_TILING_OPTIMAL**, and **usage** containing **VK_IMAGE_USAGE_STORAGE_BIT**.

- **maxSampleMaskWords** is the maximum number of array elements of a variable decorated with the **SampleMask** built-in decoration.

- **timestampComputeAndGraphics** specifies support for timestamps on all graphics and compute queues. If this limit is set to **VK_TRUE**, all queues that advertise the **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT** in the **VkQueueFamilyProperties::queueFlags** support **VkQueueFamilyProperties::timestampValidBits** of at least 36. See **Timestamp Queries**.

- **timestampPeriod** is the number of nanoseconds **required** for a timestamp query to be incremented by 1. See **Timestamp Queries**.

- **maxClipDistances** is the maximum number of clip distances that can be used in a single shader stage. The size of any array declared with the **ClipDistance** built-in decoration in a shader module must be less than or equal to this limit.

- **maxCullDistances** is the maximum number of cull distances that can be used in a single shader stage. The size of any array declared with the **CullDistance** built-in decoration in a shader module must be less than or equal to this limit.

- **maxCombinedClipAndCullDistances** is the maximum combined number of clip and cull distances
that can be used in a single shader stage. The sum of the sizes of any pair of arrays declared with the ClipDistance and CullDistance built-in decoration used by a single shader stage in a shader module must be less than or equal to this limit.

- **discreteQueuePriorities** is the number of discrete priorities that can be assigned to a queue based on the value of each member of VkDeviceQueueCreateInfo::pQueuePriorities. This must be at least 2, and levels must be spread evenly over the range, with at least one level at 1.0, and another at 0.0. See Queue Priority.

- **pointSizeRange[2]** is the range [minimum,maximum] of supported sizes for points. Values written to variables decorated with the PointSize built-in decoration are clamped to this range.

- **lineWidthRange[2]** is the range [minimum,maximum] of supported widths for lines. Values specified by the lineWidth member of the VkPipelineRasterizationStateCreateInfo or the lineWidth parameter to vkCmdSetLineWidth are clamped to this range.

- **pointSizeGranularity** is the granularity of supported point sizes. Not all point sizes in the range defined by pointSizeRange are supported. This limit specifies the granularity (or increment) between successive supported point sizes.

- **lineWidthGranularity** is the granularity of supported line widths. Not all line widths in the range defined by lineWidthRange are supported. This limit specifies the granularity (or increment) between successive supported line widths.

- **strictLines** specifies whether lines are rasterized according to the preferred method of rasterization. If set to VK_FALSE, lines may be rasterized under a relaxed set of rules. If set to VK_TRUE, lines are rasterized as per the strict definition. See Basic Line Segment Rasterization.

- **standardSampleLocations** specifies whether rasterization uses the standard sample locations as documented in Multisampling. If set to VK_TRUE, the implementation uses the documented sample locations. If set to VK_FALSE, the implementation may use different sample locations.

- **optimalBufferCopyOffsetAlignment** is the optimal buffer offset alignment in bytes for vkCmdCopyBufferToImage2KHR, vkCmdCopyBufferToImage, vkCmdCopyImageToBuffer2KHR, and vkCmdCopyImageToBuffer. The per texel alignment requirements are enforced, but applications should use the optimal alignment for optimal performance and power use.

- **optimalBufferCopyRowPitchAlignment** is the optimal buffer row pitch alignment in bytes for vkCmdCopyBufferToImage2KHR, vkCmdCopyBufferToImage, vkCmdCopyImageToBuffer2KHR, and vkCmdCopyImageToBuffer. Row pitch is the number of bytes between texels with the same X coordinate in adjacent rows (Y coordinates differ by one). The per texel alignment requirements are enforced, but applications should use the optimal alignment for optimal performance and power use.

- **nonCoherentAtomSize** is the size and alignment in bytes that bounds concurrent access to host-mapped device memory.

For all bitmasks of VkSampleCountFlagBits, the sample count limits defined above represent the minimum supported sample counts for each image type. Individual images may support additional sample counts, which are queried using vkGetPhysicalDeviceImageFormatProperties as described in Supported Sample Counts.

Bits which may be set in the sample count limits returned by VkPhysicalDeviceLimits, as well as in
other queries and structures representing image sample counts, are:

```cpp
// Provided by VK_VERSION_1_0
typedef enum VkSampleCountFlagBits {
    VK_SAMPLE_COUNT_1_BIT = 0x00000001,
    VK_SAMPLE_COUNT_2_BIT = 0x00000002,
    VK_SAMPLE_COUNT_4_BIT = 0x00000004,
    VK_SAMPLE_COUNT_8_BIT = 0x00000008,
    VK_SAMPLE_COUNT_16_BIT = 0x00000010,
    VK_SAMPLE_COUNT_32_BIT = 0x00000020,
    VK_SAMPLE_COUNT_64_BIT = 0x00000040,
} VkSampleCountFlagBits;
```

- **VK_SAMPLE_COUNT_1_BIT** specifies an image with one sample per pixel.
- **VK_SAMPLE_COUNT_2_BIT** specifies an image with 2 samples per pixel.
- **VK_SAMPLE_COUNT_4_BIT** specifies an image with 4 samples per pixel.
- **VK_SAMPLE_COUNT_8_BIT** specifies an image with 8 samples per pixel.
- **VK_SAMPLE_COUNT_16_BIT** specifies an image with 16 samples per pixel.
- **VK_SAMPLE_COUNT_32_BIT** specifies an image with 32 samples per pixel.
- **VK_SAMPLE_COUNT_64_BIT** specifies an image with 64 samples per pixel.

```cpp
// Provided by VK_VERSION_1_0
typedef VkFlags VkSampleCountFlags;
```

**VkSampleCountFlags** is a bitmask type for setting a mask of zero or more **VkSampleCountFlagBits**.

The **VkPhysicalDevicePushDescriptorPropertiesKHR** structure is defined as:

```cpp
// Provided by VK_KHR_push_descriptor
typedef struct VkPhysicalDevicePushDescriptorPropertiesKHR {
    VkStructureType sType;
    void* pNext;
    uint32_t maxPushDescriptors;
} VkPhysicalDevicePushDescriptorPropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxPushDescriptors** is the maximum number of descriptors that can be used in a descriptor set created with **VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR** set.

If the **VkPhysicalDevicePushDescriptorPropertiesKHR** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.
Valid Usage (Implicit)

- VUID-VkPhysicalDevicePushDescriptorPropertiesKHR-sType-sType
  
sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PUSH_DESCRIPTOR_PROPERTIES_KHR

The `VkPhysicalDeviceMultiviewProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMultiviewProperties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxMultiviewViewCount;
    uint32_t           maxMultiviewInstanceIndex;
} VkPhysicalDeviceMultiviewProperties;
```

or the equivalent

```c
// Provided by VK_KHR_multiview
typedef VkPhysicalDeviceMultiviewProperties VkPhysicalDeviceMultiviewPropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxMultiviewViewCount** is one greater than the maximum view index that can be used in a subpass.
- **maxMultiviewInstanceIndex** is the maximum valid value of instance index allowed to be generated by a drawing command recorded within a subpass of a multiview render pass instance.

If the `VkPhysicalDeviceMultiviewProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceMultiviewProperties-sType-sType
  
sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES

The `VkPhysicalDeviceFloatControlsProperties` structure is defined as:

---

2198 | Chapter 42. Limits
```c
typedef struct VkPhysicalDeviceFloatControlsProperties {
    VkStructureType sType;
    void* pNext;
    VkShaderFloatControlsIndependence denormBehaviorIndependence;
    VkShaderFloatControlsIndependence roundingModeIndependence;
    VkBool32 shaderSignedZeroInfNanPreserveFloat16;
    VkBool32 shaderSignedZeroInfNanPreserveFloat32;
    VkBool32 shaderSignedZeroInfNanPreserveFloat64;
    VkBool32 shaderDenormPreserveFloat16;
    VkBool32 shaderDenormPreserveFloat32;
    VkBool32 shaderDenormPreserveFloat64;
    VkBool32 shaderDenormFlushToZeroFloat16;
    VkBool32 shaderDenormFlushToZeroFloat32;
    VkBool32 shaderDenormFlushToZeroFloat64;
    VkBool32 shaderRoundingModeRTEFloat16;
    VkBool32 shaderRoundingModeRTEFloat32;
    VkBool32 shaderRoundingModeRTZFloat16;
    VkBool32 shaderRoundingModeRTZFloat32;
    VkBool32 shaderRoundingModeRTZFloat64;
} VkPhysicalDeviceFloatControlsProperties;
```

or the equivalent

```c
// Provided by VK_KHR_shader_float_controls
typedef VkPhysicalDeviceFloatControlsProperties
    VkPhysicalDeviceFloatControlsPropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **denormBehaviorIndependence** is a `VkShaderFloatControlsIndependence` value indicating whether, and how, denorm behavior can be set independently for different bit widths.
- **roundingModeIndependence** is a `VkShaderFloatControlsIndependence` value indicating whether, and how, rounding modes can be set independently for different bit widths.
- **shaderSignedZeroInfNanPreserveFloat16** is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 16-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 16-bit floating-point types.
- **shaderSignedZeroInfNanPreserveFloat32** is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 32-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 32-bit floating-point types.
- **shaderSignedZeroInfNanPreserveFloat64** is a boolean value indicating whether sign of a zero, Nans and ±∞ can be preserved in 64-bit floating-point computations. It also indicates whether the `SignedZeroInfNanPreserve` execution mode can be used for 64-bit floating-point types.
- **shaderDenormPreserveFloat16** is a boolean value indicating whether denormals can be preserved...
in 16-bit floating-point computations. It also indicates whether the DenormPreserve execution mode can be used for 16-bit floating-point types.

- shaderDenormPreserveFloat32 is a boolean value indicating whether denormals can be preserved in 32-bit floating-point computations. It also indicates whether the DenormPreserve execution mode can be used for 32-bit floating-point types.

- shaderDenormPreserveFloat64 is a boolean value indicating whether denormals can be preserved in 64-bit floating-point computations. It also indicates whether the DenormPreserve execution mode can be used for 64-bit floating-point types.

- shaderDenormFlushToZeroFloat16 is a boolean value indicating whether denormals can be flushed to zero in 16-bit floating-point computations. It also indicates whether the DenormFlushToZero execution mode can be used for 16-bit floating-point types.

- shaderDenormFlushToZeroFloat32 is a boolean value indicating whether denormals can be flushed to zero in 32-bit floating-point computations. It also indicates whether the DenormFlushToZero execution mode can be used for 32-bit floating-point types.

- shaderDenormFlushToZeroFloat64 is a boolean value indicating whether denormals can be flushed to zero in 64-bit floating-point computations. It also indicates whether the DenormFlushToZero execution mode can be used for 64-bit floating-point types.

- shaderRoundingModeRTEFloat16 is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTE execution mode can be used for 16-bit floating-point types.

- shaderRoundingModeRTEFloat32 is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTE execution mode can be used for 32-bit floating-point types.

- shaderRoundingModeRTEFloat64 is a boolean value indicating whether an implementation supports the round-to-nearest-even rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTE execution mode can be used for 64-bit floating-point types.

- shaderRoundingModeRTZFloat16 is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 16-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTZ execution mode can be used for 16-bit floating-point types.

- shaderRoundingModeRTZFloat32 is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 32-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTZ execution mode can be used for 32-bit floating-point types.

- shaderRoundingModeRTZFloat64 is a boolean value indicating whether an implementation supports the round-towards-zero rounding mode for 64-bit floating-point arithmetic and conversion instructions. It also indicates whether the RoundingModeRTZ execution mode can be used for 64-bit floating-point types.

If the VkPhysicalDeviceFloatControlsProperties structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with
each corresponding implementation-dependent property.

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceFloatControlsProperties-sType-sType**
  
  *sType* must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES**

Values which *may* be returned in the `denormBehaviorIndependence` and `roundingModeIndependence` fields of `VkPhysicalDeviceFloatControlsProperties` are:

```c
typedef enum VkShaderFloatControlsIndependence {
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY = 0,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL = 1,
    VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE = 2,
} VkShaderFloatControlsIndependence;
```

or the equivalent

```c
// Provided by VK_KHR_shader_float_controls
typedef VkShaderFloatControlsIndependence VkShaderFloatControlsIndependenceKHR;
```

- **VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY** specifies that shader float controls for 32-bit floating point can be set independently; other bit widths must be set identically to each other.

- **VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL** specifies that shader float controls for all bit widths can be set independently.

- **VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE** specifies that shader float controls for all bit widths must be set identically.

The `VkPhysicalDeviceDiscardRectanglePropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_discard_rectangles
typedef struct VkPhysicalDeviceDiscardRectanglePropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxDiscardRectangles;
} VkPhysicalDeviceDiscardRectanglePropertiesEXT;
```
• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **maxDiscardRectangles** is the maximum number of active discard rectangles that can be specified.

If the `VkPhysicalDeviceDiscardRectanglePropertiesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceDiscardRectanglePropertiesEXT-sType-sType
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT**

The `VkPhysicalDeviceSampleLocationsPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_sample_locations
typedef struct VkPhysicalDeviceSampleLocationsPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkSampleCountFlags sampleLocationSampleCounts;
    VkExtent2D maxSampleLocationGridSize;
    float sampleLocationCoordinateRange[2];
    uint32_t sampleLocationSubPixelBits;
    VkBool32 variableSampleLocations;
} VkPhysicalDeviceSampleLocationsPropertiesEXT;
```

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **sampleLocationSampleCounts** is a bitmask of `VkSampleCountFlagBits` indicating the sample counts supporting custom sample locations.

• **maxSampleLocationGridSize** is the maximum size of the pixel grid in which sample locations can vary that is supported for all sample counts in `sampleLocationSampleCounts`.

• **sampleLocationCoordinateRange[2]** is the range of supported sample location coordinates.

• **sampleLocationSubPixelBits** is the number of bits of subpixel precision for sample locations.

• **variableSampleLocations** specifies whether the sample locations used by all pipelines that will be bound to a command buffer during a subpass must match. If set to **VK_TRUE**, the implementation supports variable sample locations in a subpass. If set to **VK_FALSE**, then the sample locations must stay constant in each subpass.

If the `VkPhysicalDeviceSampleLocationsPropertiesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.
Valid Usage (Implicit)

• VUID-VkPhysicalDeviceSampleLocationsPropertiesEXT-sType-sType

sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT

The VkPhysicalDeviceExternalMemoryHostPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_external_memory_host
typedef struct VkPhysicalDeviceExternalMemoryHostPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkDeviceSize       minImportedHostPointerAlignment;
} VkPhysicalDeviceExternalMemoryHostPropertiesEXT;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• minImportedHostPointerAlignment is the minimum required alignment, in bytes, for the base address and size of host pointers that can be imported to a Vulkan memory object.

If the VkPhysicalDeviceExternalMemoryHostPropertiesEXT structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

• VUID-VkPhysicalDeviceExternalMemoryHostPropertiesEXT-sType-sType

sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_MEMORY_HOST_PROPERTIES_EXT

The VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX structure is defined as:

```c
// Provided by VK_NVX_multiview_per_view_attributes
typedef struct VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           perViewPositionAllComponents;
} VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• perViewPositionAllComponents is VK_TRUE if the implementation supports per-view position values that differ in components other than the X component.

If the VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is...
filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PER_VIEW_ATTRIBUTES_PROPERTIES_NVX`.

The `VkPhysicalDeviceMultiviewPerViewAttributesProperties` structure is defined as:

```c
typedef struct VkPhysicalDeviceMultiviewPerViewAttributesProperties {
    VkStructureType sType;
    void* pNext;
    VkMultiviewPerViewAttributesKindPerViewKindCountKHR multiviewPerViewAttributes;
} VkPhysicalDeviceMultiviewPerViewAttributesProperties;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance2
typedef VkPhysicalDeviceMultiviewPerViewAttributesProperties
    VkPhysicalDeviceMultiviewPerViewAttributesPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pointClippingBehavior` is a `VkPointClippingBehavior` value specifying the point clipping behavior supported by the implementation.

If the `VkPhysicalDeviceMultiviewPerViewAttributesProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDevicePointClippingProperties-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES`.

The `VkPhysicalDevicePointClippingProperties` structure is defined as:

```c
typedef struct VkPhysicalDevicePointClippingProperties {
    VkStructureType sType;
    void* pNext;
    VkPointClippingBehavior pointClippingBehavior;
} VkPhysicalDevicePointClippingProperties;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance2
typedef VkPhysicalDevicePointClippingProperties
    VkPhysicalDevicePointClippingPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `pointClippingBehavior` is a `VkPointClippingBehavior` value specifying the point clipping behavior supported by the implementation.

If the `VkPhysicalDevicePointClippingProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceSubgroupProperties-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES`.

The `VkPhysicalDeviceSubgroupProperties` structure is defined as:
typedef struct VkPhysicalDeviceSubgroupProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t subgroupSize;
    VkShaderStageFlags supportedStages;
    VkSubgroupFeatureFlags supportedOperations;
    VkBool32 quadOperationsInAllStages;
} VkPhysicalDeviceSubgroupProperties;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.

- **subgroupSize** is the default number of invocations in each subgroup. subgroupSize is at least 1 if any of the physical device's queues support VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT. subgroupSize is a power-of-two.

- **supportedStages** is a bitfield of VkShaderStageFlagBits describing the shader stages that group operations with subgroup scope are supported in. supportedStages will have the VK_SHADER_STAGE_COMPUTE_BIT bit set if any of the physical device's queues support VK_QUEUE_COMPUTE_BIT.

- **supportedOperations** is a bitmask of VkSubgroupFeatureFlagBits specifying the sets of group operations with subgroup scope supported on this device. supportedOperations will have the VK_SUBGROUP_FEATURE_BASIC_BIT bit set if any of the physical device's queues support VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT.

- **quadOperationsInAllStages** is a boolean specifying whether quad group operations are available in all stages, or are restricted to fragment and compute stages.

If the VkPhysicalDeviceSubgroupProperties structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

If supportedOperations includes VK_SUBGROUP_FEATURE_QUAD_BIT, subgroupSize must be greater than or equal to 4.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceSubgroupProperties-sType-sType
  
  **sType** must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_PROPERTIES

Bits which can be set in VkPhysicalDeviceSubgroupProperties::supportedOperations to specify supported group operations with subgroup scope are:
typedef enum VkSubgroupFeatureFlagBits {
    VK_SUBGROUP_FEATURE_BASIC_BIT = 0x00000001,
    VK_SUBGROUP_FEATURE_VOTE_BIT = 0x00000002,
    VK_SUBGROUP_FEATURE_ARITHMETIC_BIT = 0x00000004,
    VK_SUBGROUP_FEATURE_BALLOT_BIT = 0x00000008,
    VK_SUBGROUP_FEATURE_SHUFFLE_BIT = 0x00000010,
    VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT = 0x00000020,
    VK_SUBGROUP_FEATURE_CLUSTERED_BIT = 0x00000040,
    VK_SUBGROUP_FEATURE_QUAD_BIT = 0x00000080,
    // Provided by VK_NV_shader_subgroup_partitioned
    VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV = 0x00000100,
} VkSubgroupFeatureFlagBits;

// Provided by VK_VERSION_1_1
typedef VkFlags VkSubgroupFeatureFlags;

VkSubgroupFeatureFlags is a bitmask type for setting a mask of zero or more VkSubgroupFeatureFlagBits.

The VkPhysicalDeviceSubgroupSizeControlPropertiesEXT structure is defined as:
// Provided by VK_EXT_subgroup_size_control

typedef struct VkPhysicalDeviceSubgroupSizeControlPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t minSubgroupSize;
    uint32_t maxSubgroupSize;
    uint32_t maxComputeWorkgroupSubgroups;
    VkShaderStageFlags requiredSubgroupSizeStages;
} VkPhysicalDeviceSubgroupSizeControlPropertiesEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **minSubgroupSize** is the minimum subgroup size supported by this device. **minSubgroupSize** is at least one if any of the physical device’s queues support **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**. **minSubgroupSize** is a power-of-two. **minSubgroupSize** is less than or equal to **maxSubgroupSize**. **minSubgroupSize** is less than or equal to **subgroupSize**.
- **maxSubgroupSize** is the maximum subgroup size supported by this device. **maxSubgroupSize** is at least one if any of the physical device’s queues support **VK_QUEUE_GRAPHICS_BIT** or **VK_QUEUE_COMPUTE_BIT**. **maxSubgroupSize** is a power-of-two. **maxSubgroupSize** is greater than or equal to **minSubgroupSize**. **maxSubgroupSize** is greater than or equal to **subgroupSize**.
- **maxComputeWorkgroupSubgroups** is the maximum number of subgroups supported by the implementation within a workgroup.
- **requiredSubgroupSizeStages** is a bitfield of what shader stages support having a required subgroup size specified.

If the VkPhysicalDeviceSubgroupSizeControlPropertiesEXT structure is included in the **pNext** chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

If **VkPhysicalDeviceSubgroupProperties::supportedOperations** includes **VK_SUBGROUP_FEATURE_QUAD_BIT**, **minSubgroupSize** must be greater than or equal to 4.

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceSubgroupSizeControlPropertiesEXT-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT**

The VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT structure is defined as:
// Provided by VK_EXT_blend_operation_advanced
typedef struct VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           advancedBlendMaxColorAttachments;
    VkBool32           advancedBlendIndependentBlend;
    VkBool32           advancedBlendNonPremultipliedSrcColor;
    VkBool32           advancedBlendNonPremultipliedDstColor;
    VkBool32           advancedBlendCorrelatedOverlap;
    VkBool32           advancedBlendAllOperations;
} VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT;

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **advancedBlendMaxColorAttachments** is one greater than the highest color attachment index that can be used in a subpass, for a pipeline that uses an advanced blend operation.

• **advancedBlendIndependentBlend** specifies whether advanced blend operations can vary per-attachment.

• **advancedBlendNonPremultipliedSrcColor** specifies whether the source color can be treated as non-premultiplied. If this is **VK_FALSE**, then VkPipelineColorBlendAdvancedStateCreateInfoEXT ::srcPremultiplied must be **VK_TRUE**.

• **advancedBlendNonPremultipliedDstColor** specifies whether the destination color can be treated as non-premultiplied. If this is **VK_FALSE**, then VkPipelineColorBlendAdvancedStateCreateInfoEXT ::dstPremultiplied must be **VK_TRUE**.

• **advancedBlendCorrelatedOverlap** specifies whether the overlap mode can be treated as correlated. If this is **VK_FALSE**, then VkPipelineColorBlendAdvancedStateCreateInfoEXT ::blendOverlap must be **VK_BLEND_OVERLAP_UNCORRELATED_EXT**.

• **advancedBlendAllOperations** specifies whether all advanced blend operation enums are supported. See the valid usage of VkPipelineColorBlendAttachmentState.

If the VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT structure is included in the **pNext** chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

---

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT-sType-sType**
  
  
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT**

The VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT structure is defined as:
typedef struct VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxVertexAttribDivisor;
} VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT;

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **maxVertexAttribDivisor** is the maximum value of the number of instances that will repeat the value of vertex attribute data when instanced rendering is enabled.

If the `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT-sType-sType

The `VkPhysicalDeviceSamplerFilterMinmaxProperties` structure is defined as:

typedef struct VkPhysicalDeviceSamplerFilterMinmaxProperties {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           filterMinmaxSingleComponentFormats;
    VkBool32           filterMinmaxImageComponentMapping;
} VkPhysicalDeviceSamplerFilterMinmaxProperties;

or the equivalent

```
// Provided by VK_EXT_sampler_filter_minmax
typedef VkPhysicalDeviceSamplerFilterMinmaxProperties
VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **filterMinmaxSingleComponentFormats** is a boolean value indicating whether a minimum set of required formats support min/max filtering.
- **filterMinmaxImageComponentMapping** is a boolean value indicating whether the implementation supports non-identity component mapping of the image when doing min/max filtering.
If the `VkPhysicalDeviceSamplerFilterMinmaxProperties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

If `filterMinmaxSingleComponentFormats` is `VK_TRUE`, the following formats **must** support the `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT` feature with `VK_IMAGE_TILING_OPTIMAL`, if they support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`:

- `VK_FORMAT_R8_UNORM`
- `VK_FORMAT_R8_SNORM`
- `VK_FORMAT_R16_UNORM`
- `VK_FORMAT_R16_SNORM`
- `VK_FORMAT_R16_SFLOAT`
- `VK_FORMAT_R32_SFLOAT`
- `VK_FORMAT_D16_UNORM`
- `VK_FORMAT_X8_D24_UNORM_PACK32`
- `VK_FORMAT_D32_SFLOAT`
- `VK_FORMAT_D16_UNORM_S8_UINT`
- `VK_FORMAT_D24_UNORM_S8_UINT`
- `VK_FORMAT_D32_SFLOAT_S8_UINT`

If the format is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports min/max filtering, and that min/max filtering of the depth aspect is supported when depth compare is disabled in the sampler.

If `filterMinmaxImageComponentMapping` is `VK_FALSE` the component mapping of the image view used with min/max filtering **must** have been created with the `r` component set to the identity swizzle. Only the `r` component of the sampled image value is defined and the other component values are undefined. If `filterMinmaxImageComponentMapping` is `VK_TRUE` this restriction does not apply and image component mapping works as normal.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceSamplerFilterMinmaxProperties-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES`

The `VkPhysicalDeviceProtectedMemoryProperties` structure is defined as:
typedef struct VkPhysicalDeviceProtectedMemoryProperties {
    VkStructureType    sType;
    void*              pNext;
    VkBool32           protectedNoFault;
} VkPhysicalDeviceProtectedMemoryProperties;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **protectedNoFault** specifies the behavior of the implementation when protected memory access rules are broken. If protectedNoFault is VK_TRUE, breaking those rules will not result in process termination or device loss.

If the `VkPhysicalDeviceProtectedMemoryProperties` structure is included in the **pNext** chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceProtectedMemoryProperties-sType-sType**
  - **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROTECTED_MEMORY_PROPERTIES**

The `VkPhysicalDeviceMaintenance3Properties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceMaintenance3Properties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxPerSetDescriptors;
    VkDeviceSize       maxMemoryAllocationSize;
} VkPhysicalDeviceMaintenance3Properties;
```

or the equivalent

```c
// Provided by VK_KHR_maintenance3
typedef VkPhysicalDeviceMaintenance3Properties
VkPhysicalDeviceMaintenance3PropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxPerSetDescriptors** is a maximum number of descriptors (summed over all descriptor types) in a single descriptor set that is guaranteed to satisfy any implementation-dependent constraints on the size of a descriptor set itself. Applications **can** query whether a descriptor set
that goes beyond this limit is supported using `vkGetDescriptorSetLayoutSupport`.

- `maxMemoryAllocationSize` is the maximum size of a memory allocation that can be created, even if there is more space available in the heap.

If the `VkPhysicalDeviceMaintenance3Properties` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceMaintenance3Properties-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES`

The `VkPhysicalDeviceMeshShaderPropertiesNV` structure is defined as:

```c
// Provided by VK_NV_mesh_shader
typedef struct VkPhysicalDeviceMeshShaderPropertiesNV {
  VkStructureType sType;
  void* pNext;
  uint32_t maxDrawMeshTasksCount;
  uint32_t maxTaskWorkGroupInvocations;
  uint32_t maxTaskWorkGroupSize[3];
  uint32_t maxTaskTotalMemorySize;
  uint32_t maxTaskOutputCount;
  uint32_t maxMeshWorkGroupInvocations;
  uint32_t maxMeshWorkGroupSize[3];
  uint32_t maxMeshTotalMemorySize;
  uint32_t maxMeshOutputVertices;
  uint32_t maxMeshOutputPrimitives;
  uint32_t maxMeshMultiviewViewCount;
  uint32_t meshOutputPerVertexGranularity;
  uint32_t meshOutputPerPrimitiveGranularity;
} VkPhysicalDeviceMeshShaderPropertiesNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxDrawMeshTasksCount` is the maximum number of local workgroups that can be launched by a single draw mesh tasks command. See Programmable Mesh Shading.
- `maxTaskWorkGroupInvocations` is the maximum total number of task shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the `LocalSize` execution mode in shader modules or by the object decorated by the `WorkgroupSize` decoration, must be less than or equal to this limit.
- `maxTaskWorkGroupSize[3]` is the maximum size of a local task workgroup. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The x, y, and z sizes, as specified by the `LocalSize` execution mode or by the object decorated by the
WorkgroupSize decoration in shader modules, **must** be less than or equal to the corresponding limit.

- **maxTaskTotalMemorySize** is the maximum number of bytes that the task shader can use in total for shared and output memory combined.

- **maxTaskOutputCount** is the maximum number of output tasks a single task shader workgroup can emit.

- **maxMeshWorkGroupInvocations** is the maximum total number of mesh shader invocations in a single local workgroup. The product of the X, Y, and Z sizes, as specified by the LocalSize execution mode in shader modules or by the object decorated by the WorkgroupSize decoration, **must** be less than or equal to this limit.

- **maxMeshWorkGroupSize[3]** is the maximum size of a local mesh workgroup. These three values represent the maximum local workgroup size in the X, Y, and Z dimensions, respectively. The x, y, and z sizes, as specified by the LocalSize execution mode or by the object decorated by the WorkgroupSize decoration in shader modules, **must** be less than or equal to the corresponding limit.

- **maxMeshTotalMemorySize** is the maximum number of bytes that the mesh shader can use in total for shared and output memory combined.

- **maxMeshOutputVertices** is the maximum number of vertices a mesh shader output can store.

- **maxMeshOutputPrimitives** is the maximum number of primitives a mesh shader output can store.

- **maxMeshMultiviewViewCount** is the maximum number of multi-view views a mesh shader can use.

- **meshOutputPerVertexGranularity** is the granularity with which mesh vertex outputs are allocated. The value can be used to compute the memory size used by the mesh shader, which must be less than or equal to maxMeshTotalMemorySize.

- **meshOutputPerPrimitiveGranularity** is the granularity with which mesh outputs qualified as per-primitive are allocated. The value can be used to compute the memory size used by the mesh shader, which must be less than or equal to maxMeshTotalMemorySize.

If the **VkPhysicalDeviceMeshShaderPropertiesNV** structure is included in the pNext chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceMeshShaderPropertiesNV-sType-sType**
  
  sType **must** be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_PROPERTIES_NV**

The **VkPhysicalDeviceDescriptorIndexingProperties** structure is defined as:
typedef struct VkPhysicalDeviceDescriptorIndexingProperties {
    VkStructureType sType;
    void* pNext;
    uint32_t maxUpdateAfterBindDescriptorsInAllPools;
    VkBool32 shaderUniformBufferArrayNonUniformIndexingNative;
    VkBool32 shaderSampledImageArrayNonUniformIndexingNative;
    VkBool32 shaderStorageBufferArrayNonUniformIndexingNative;
    VkBool32 shaderStorageImageArrayNonUniformIndexingNative;
    VkBool32 shaderInputAttachmentArrayNonUniformIndexingNative;
    VkBool32 robustBufferAccessUpdateAfterBind;
    VkBool32 quadDivergentImplicitLod;
    uint32_t maxPerStageDescriptorUpdateAfterBindSamplers;
    uint32_t maxPerStageDescriptorUpdateAfterBindUniformBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageBuffers;
    uint32_t maxPerStageDescriptorUpdateAfterBindSampledImages;
    uint32_t maxPerStageDescriptorUpdateAfterBindStorageImages;
    uint32_t maxPerStageDescriptorUpdateAfterBindInputAttachments;
    uint32_t maxDescriptorSetUpdateAfterBindSamplers;
    uint32_t maxDescriptorSetUpdateAfterBindUniformBuffers;
    uint32_t maxDescriptorSetUpdateAfterBindUniformBuffersDynamic;
    uint32_t maxDescriptorSetUpdateAfterBindStorageBuffers;
    uint32_t maxDescriptorSetUpdateAfterBindStorageBuffersDynamic;
    uint32_t maxDescriptorSetUpdateAfterBindSampledImages;
    uint32_t maxDescriptorSetUpdateAfterBindStorageImages;
    uint32_t maxDescriptorSetUpdateAfterBindInputAttachments;
} VkPhysicalDeviceDescriptorIndexingProperties;

or the equivalent

// Provided by VK_EXT_descriptor_indexing
typedef VkPhysicalDeviceDescriptorIndexingProperties
    VkPhysicalDeviceDescriptorIndexingPropertiesEXT;

• sType is the type of this structure.
• pNext is NULL or a pointer to a structure extending this structure.
• maxUpdateAfterBindDescriptorsInAllPools is the maximum number of descriptors (summed over all descriptor types) that can be created across all pools that are created with the VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT bit set. Pool creation may fail when this limit is exceeded, or when the space this limit represents is unable to satisfy a pool creation due to fragmentation.
• shaderUniformBufferArrayNonUniformIndexingNative is a boolean value indicating whether uniform buffer descriptors natively support nonuniform indexing. If this is VK_FALSE, then a single dynamic instance of an instruction that nonuniformly indexes an array of uniform buffers may execute multiple times in order to access all the descriptors.
• shaderSampledImageArrayNonUniformIndexingNative is a boolean value indicating whether
sampler and image descriptors natively support nonuniform indexing. If this is VK_FALSE, then a single dynamic instance of an instruction that nonuniformly indexes an array of samplers or images may execute multiple times in order to access all the descriptors.

- **shaderStorageBufferArrayNonUniformIndexingNative** is a boolean value indicating whether storage buffer descriptors natively support nonuniform indexing. If this is VK_FALSE, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage buffers may execute multiple times in order to access all the descriptors.

- **shaderStorageImageArrayNonUniformIndexingNative** is a boolean value indicating whether storage image descriptors natively support nonuniform indexing. If this is VK_FALSE, then a single dynamic instance of an instruction that nonuniformly indexes an array of storage images may execute multiple times in order to access all the descriptors.

- **shaderInputAttachmentArrayNonUniformIndexingNative** is a boolean value indicating whether input attachment descriptors natively support nonuniform indexing. If this is VK_FALSE, then a single dynamic instance of an instruction that nonuniformly indexes an array of input attachments may execute multiple times in order to access all the descriptors.

- **robustBufferAccessUpdateAfterBind** is a boolean value indicating whether robustBufferAccess can be enabled in a device simultaneously with descriptorBindingUniformBufferUpdateAfterBind, descriptorBindingStorageBufferUpdateAfterBind, descriptorBindingUniformTexelBufferUpdateAfterBind, and/or descriptorBindingStorageTexelBufferUpdateAfterBind. If this is VK_FALSE, then either robustBufferAccess must be disabled or all of these update-after-bind features must be disabled.

- **quadDivergentImplicitLod** is a boolean value indicating whether implicit level of detail calculations for image operations have well-defined results when the image and/or sampler objects used for the instruction are not uniform within a quad. See Derivative Image Operations.

- **maxPerStageDescriptorUpdateAfterBindSamplers** is similar to maxPerStageDescriptorSamplers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- **maxPerStageDescriptorUpdateAfterBindUniformBuffers** is similar to maxPerStageDescriptorUniformBuffers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- **maxPerStageDescriptorUpdateAfterBindStorageBuffers** is similar to maxPerStageDescriptorStorageBuffers but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- **maxPerStageDescriptorUpdateAfterBindSampledImages** is similar to maxPerStageDescriptorSampledImages but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- **maxPerStageDescriptorUpdateAfterBindStorageImages** is similar to maxPerStageDescriptorStorageImages but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

- **maxPerStageDescriptorUpdateAfterBindInputAttachments** is similar to maxPerStageDescriptorInputAttachments but counts descriptors from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.
• **maxPerStageUpdateAfterBindResources** is similar to **maxPerStageResources** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

• **maxDescriptorSetUpdateAfterBindSamplers** is similar to **maxDescriptorSetSamplers** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

• **maxDescriptorSetUpdateAfterBindUniformBuffers** is similar to **maxDescriptorSetUniformBuffers** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

• **maxDescriptorSetUpdateAfterBindUniformBuffersDynamic** is similar to **maxDescriptorSetUniformBuffersDynamic** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set. While an application can allocate dynamic uniform buffer descriptors from a pool created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT`, bindings for these descriptors must not be present in any descriptor set layout that includes bindings created with `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

• **maxDescriptorSetUpdateAfterBindStorageBuffers** is similar to **maxDescriptorSetStorageBuffers** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

• **maxDescriptorSetUpdateAfterBindStorageBuffersDynamic** is similar to **maxDescriptorSetStorageBuffersDynamic** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set. While an application can allocate dynamic storage buffer descriptors from a pool created with the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT`, bindings for these descriptors must not be present in any descriptor set layout that includes bindings created with `VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT`.

• **maxDescriptorSetUpdateAfterBindSampledImages** is similar to **maxDescriptorSetSampledImages** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

• **maxDescriptorSetUpdateAfterBindStorageImages** is similar to **maxDescriptorSetStorageImages** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

• **maxDescriptorSetUpdateAfterBindInputAttachments** is similar to **maxDescriptorSetInputAttachments** but counts descriptors from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

If the **VkPhysicalDeviceDescriptorIndexingProperties** structure is included in the `pNext` chain of the **VkPhysicalDeviceProperties2** structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

---

**Valid Usage (Implicit)**

- **VUID-VkPhysicalDeviceDescriptorIndexingProperties-sType-sType**

  `sType` must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_PROPERTIES**
The VkPhysicalDeviceInlineUniformBlockPropertiesEXT structure is defined as:

```c
// Provided by VK_EXT_inline_uniform_block
typedef struct VkPhysicalDeviceInlineUniformBlockPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxInlineUniformBlockSize;
    uint32_t           maxPerStageDescriptorInlineUniformBlocks;
    uint32_t           maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks;
    uint32_t           maxDescriptorSetInlineUniformBlocks;
    uint32_t           maxDescriptorSetUpdateAfterBindInlineUniformBlocks;
} VkPhysicalDeviceInlineUniformBlockPropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxInlineUniformBlockSize** is the maximum size in bytes of an inline uniform block binding.
- **maxPerStageDescriptorInlineUniformBlock** is the maximum number of inline uniform block bindings that can be accessible to a single shader stage in a pipeline layout. Descriptor bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT count against this limit. Only descriptor bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit.
- **maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks** is similar to maxPerStageDescriptorInlineUniformBlocks but counts descriptor bindings from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.
- **maxDescriptorSetInlineUniformBlocks** is the maximum number of inline uniform block bindings that can be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptor bindings with a descriptor type of VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT count against this limit. Only descriptor bindings in descriptor set layouts created without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set count against this limit.
- **maxDescriptorSetUpdateAfterBindInlineUniformBlocks** is similar to maxDescriptorSetInlineUniformBlocks but counts descriptor bindings from descriptor sets created with or without the VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT bit set.

If the VkPhysicalDeviceInlineUniformBlockPropertiesEXT structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceInlineUniformBlockPropertiesEXT-sType-sType**
  
  *sType* must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES_EXT

The VkPhysicalDeviceConservativeRasterizationPropertiesEXT structure is defined as:
typedef struct VkPhysicalDeviceConservativeRasterizationPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    float              primitiveOverestimationSize;
    float              maxExtraPrimitiveOverestimationSize;
    float              extraPrimitiveOverestimationSizeGranularity;
    VkBool32           primitiveUnderestimation;
    VkBool32           conservativePointAndLineRasterization;
    VkBool32           degenerateTrianglesRasterized;
    VkBool32           degenerateLinesRasterized;
    VkBool32           fullyCoveredFragmentShaderInputVariable;
    VkBool32           conservativeRasterizationPostDepthCoverage;
} VkPhysicalDeviceConservativeRasterizationPropertiesEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **primitiveOverestimationSize** is the size in pixels the generating primitive is increased at each of its edges during conservative rasterization overestimation mode. Even with a size of 0.0, conservative rasterization overestimation rules still apply and if any part of the pixel rectangle is covered by the generating primitive, fragments are generated for the entire pixel. However implementations may make the pixel coverage area even more conservative by increasing the size of the generating primitive.
- **maxExtraPrimitiveOverestimationSize** is the maximum size in pixels of extra overestimation the implementation supports in the pipeline state. A value of 0.0 means the implementation does not support any additional overestimation of the generating primitive during conservative rasterization. A value above 0.0 allows the application to further increase the size of the generating primitive during conservative rasterization overestimation.
- **extraPrimitiveOverestimationSizeGranularity** is the granularity of extra overestimation that can be specified in the pipeline state between 0.0 and **maxExtraPrimitiveOverestimationSize** inclusive. A value of 0.0 means the implementation can use the smallest representable non-zero value in the screen space pixel fixed-point grid.
- **primitiveUnderestimation** is true if the implementation supports the `VK_CONSERVATIVE_RASTERIZATION_MODE_UNDERESTIMATE_EXT` conservative rasterization mode in addition to `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT`. Otherwise the implementation only supports `VK_CONSERVATIVE_RASTERIZATION_MODE_OVERESTIMATE_EXT`.
- **conservativePointAndLineRasterization** is true if the implementation supports conservative rasterization of point and line primitives as well as triangle primitives. Otherwise the implementation only supports triangle primitives.
- **degenerateTrianglesRasterized** is false if the implementation culls primitives generated from triangles that become zero area after they are quantized to the fixed-point rasterization pixel grid. **degenerateTrianglesRasterized** is true if these primitives are not culled and the provoking vertex attributes and depth value are used for the fragments. The primitive area calculation is done on the primitive generated from the clipped triangle if applicable. Zero area primitives are backfacing and the application can enable backface culling if desired.
• **degenerateLinesRasterized** is false if the implementation culls lines that become zero length after they are quantized to the fixed-point rasterization pixel grid. **degenerateLinesRasterized** is true if zero length lines are not culled and the provoking vertex attributes and depth value are used for the fragments.

• **fullyCoveredFragmentShaderInputVariable** is true if the implementation supports the SPIR-V builtin fragment shader input variable **FullyCoveredEXT** which specifies that conservative rasterization is enabled and the fragment area is fully covered by the generating primitive.

• **conservativeRasterizationPostDepthCoverage** is true if the implementation supports conservative rasterization with the **PostDepthCoverage** execution mode enabled. When supported the **SampleMask** built-in input variable will reflect the coverage after the early per-fragment depth and stencil tests are applied even when conservative rasterization is enabled. Otherwise **PostDepthCoverage** execution mode must not be used when conservative rasterization is enabled.

If the **VkPhysicalDeviceConservativeRasterizationPropertiesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

• **VUID-VkPhysicalDeviceConservativeRasterizationPropertiesEXT-sType-sType**

  ```
  sType
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT
  ```

The **VkPhysicalDeviceFragmentDensityMapPropertiesEXT** structure is defined as:

```c
// Provided by VK_EXT_fragment_density_map
typedef struct VkPhysicalDeviceFragmentDensityMapPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkExtent2D minFragmentDensityTexelSize;
    VkExtent2D maxFragmentDensityTexelSize;
    VkBool32 fragmentDensityInvocations;
} VkPhysicalDeviceFragmentDensityMapPropertiesEXT;
```

• **sType** is the type of this structure.

• **pNext** is **NULL** or a pointer to a structure extending this structure.

• **minFragmentDensityTexelSize** is the minimum fragment density texel size.

• **maxFragmentDensityTexelSize** is the maximum fragment density texel size.

• **fragmentDensityInvocations** specifies whether the implementation may invoke additional fragment shader invocations for each covered sample.

If the **VkPhysicalDeviceFragmentDensityMapPropertiesEXT** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceFragmentDensityMapPropertiesEXT-sType-sType**

  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_PROPERTIES_EXT`

The `VkPhysicalDeviceFragmentDensityMap2PropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_fragment_density_map2
typedef struct VkPhysicalDeviceFragmentDensityMap2PropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 subsampledLoads;
    VkBool32 subsampledCoarseReconstructionEarlyAccess;
    uint32_t maxSubsampledArrayLayers;
    uint32_t maxDescriptorSetSubsampledSamplers;
} VkPhysicalDeviceFragmentDensityMap2PropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is `NULL` or a pointer to a structure extending this structure.
- **subsampledLoads** specifies if performing image data read with load operations on subsampled attachments will be resampled to the fragment density of the render pass.
- **subsampledCoarseReconstructionEarlyAccess** specifies if performing image data read with samplers created with flags containing `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT` in fragment shader will trigger additional reads during `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`.
- **maxSubsampledArrayLayers** is the maximum number of `VkImageView` array layers for usages supporting subsampled samplers.
- **maxDescriptorSetSubsampledSamplers** is the maximum number of subsampled samplers that can be included in a `VkPipelineLayout`.

If the `VkPhysicalDeviceFragmentDensityMap2PropertiesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceFragmentDensityMap2PropertiesEXT-sType-sType**

  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_2_PROPERTIES_EXT`

The `VkPhysicalDeviceShaderCorePropertiesAMD` structure is defined as:
// Provided by VK_AMD_shader_core_properties

typedef struct VkPhysicalDeviceShaderCorePropertiesAMD {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           shaderEngineCount;
    uint32_t           shaderArraysPerEngineCount;
    uint32_t           computeUnitsPerShaderArray;
    uint32_t           simdPerComputeUnit;
    uint32_t           wavefrontsPerSimd;
    uint32_t           wavefrontSize;
    uint32_t           sgprsPerSimd;
    uint32_t           minSgprAllocation;
    uint32_t           maxSgprAllocation;
    uint32_t           sgprAllocationGranularity;
    uint32_t           vgprsPerSimd;
    uint32_t           minVgprAllocation;
    uint32_t           maxVgprAllocation;
    uint32_t           vgprAllocationGranularity;
} VkPhysicalDeviceShaderCorePropertiesAMD;

• \textit{sType} is the type of this structure.

• \textit{pNext} is NULL or a pointer to a structure extending this structure.

• \textit{shaderEngineCount} is an unsigned integer value indicating the number of shader engines found inside the shader core of the physical device.

• \textit{shaderArraysPerEngineCount} is an unsigned integer value indicating the number of shader arrays inside a shader engine. Each shader array has its own scan converter, set of compute units, and a render back end (color and depth buffers). Shader arrays within a shader engine share shader processor input (wave launcher) and shader export (export buffer) units. Currently, a shader engine can have one or two shader arrays.

• \textit{computeUnitsPerShaderArray} is an unsigned integer value indicating the physical number of compute units within a shader array. The active number of compute units in a shader array \textbf{may} be lower. A compute unit houses a set of SIMDs along with a sequencer module and a local data store.

• \textit{simdPerComputeUnit} is an unsigned integer value indicating the number of SIMDs inside a compute unit. Each SIMD processes a single instruction at a time.

• \textit{wavefrontSize} is an unsigned integer value indicating the maximum size of a subgroup.

• \textit{sgprsPerSimd} is an unsigned integer value indicating the number of physical Scalar General Purpose Registers (SGPRs) per SIMD.

• \textit{minSgprAllocation} is an unsigned integer value indicating the minimum number of SGPRs allocated for a wave.

• \textit{maxSgprAllocation} is an unsigned integer value indicating the maximum number of SGPRs allocated for a wave.

• \textit{sgprAllocationGranularity} is an unsigned integer value indicating the granularity of SGPR allocation for a wave.
• `vgprsPerSimd` is an unsigned integer value indicating the number of physical Vector General Purpose Registers (VGPRs) per SIMD.

• `minVgprAllocation` is an unsigned integer value indicating the minimum number of VGPRs allocated for a wave.

• `maxVgprAllocation` is an unsigned integer value indicating the maximum number of VGPRs allocated for a wave.

• `vgprAllocationGranularity` is an unsigned integer value indicating the granularity of VGPR allocation for a wave.

If the `VkPhysicalDeviceShaderCorePropertiesAMD` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceShaderCorePropertiesAMD-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_CORE_PROPERTIES_AMD`

The `VkPhysicalDeviceShaderCoreProperties2AMD` structure is defined as:

```c
// Provided by VK_AMD_shader_core_properties2
typedef struct VkPhysicalDeviceShaderCoreProperties2AMD {
    VkStructureType                   sType;
    void*                             pNext;
    VkShaderCorePropertiesFlagsAMD    shaderCoreFeatures;
    uint32_t                          activeComputeUnitCount;
} VkPhysicalDeviceShaderCoreProperties2AMD;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **shaderCoreFeatures** is a bitmask of `VkShaderCorePropertiesFlagBitsAMD` indicating the set of features supported by the shader core.
- **activeComputeUnitCount** is an unsigned integer value indicating the number of compute units that have been enabled.

If the `VkPhysicalDeviceShaderCoreProperties2AMD` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceShaderCoreProperties2AMD-sType-sType**
  
  *sType* must be `VK_STRUCTURE_TYPE_PHYSICALDEVICE_SHADER_CORE_PROPERTIES_2_AMD`
Bits for this type **may** be defined by future extensions, or new versions of the `VK_AMD_shader_core_properties2` extension. Possible values of the `flags` member of `VkShaderCorePropertiesFlagsAMD` are:

```c
// Provided by VK_AMD_shader_core_properties2
typedef enum VkShaderCorePropertiesFlagBitsAMD {
} VkShaderCorePropertiesFlagBitsAMD;
```

```c
// Provided by VK_AMD_shader_core_properties2
typedef VkFlags VkShaderCorePropertiesFlagsAMD;
```

`VkShaderCorePropertiesFlagsAMD` is a bitmask type for providing zero or more `VkShaderCorePropertiesFlagBitsAMD`.

The `VkPhysicalDeviceDepthStencilResolveProperties` structure is defined as:

```c
typedef struct VkPhysicalDeviceDepthStencilResolveProperties {
    VkStructureType       sType;
    void*                 pNext;
    VkResolveModeFlags    supportedDepthResolveModes;
    VkResolveModeFlags    supportedStencilResolveModes;
    VkBool32              independentResolveNone;
    VkBool32              independentResolve;
} VkPhysicalDeviceDepthStencilResolveProperties;
```

or the equivalent

```c
// Provided by VK_KHR_depth_stencil_resolve
typedef VkPhysicalDeviceDepthStencilResolveProperties
    VkPhysicalDeviceDepthStencilResolvePropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `supportedDepthResolveModes` is a bitmask of `VkResolveModeFlagBits` indicating the set of supported depth resolve modes. `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT` must be included in the set but implementations may support additional modes.
- `supportedStencilResolveModes` is a bitmask of `VkResolveModeFlagBits` indicating the set of supported stencil resolve modes. `VK_RESOLVE_MODE_SAMPLE_ZERO_BIT` must be included in the set but implementations may support additional modes. `VK_RESOLVE_MODE_AVERAGE_BIT` must not be included in the set.
- `independentResolveNone` is `VK_TRUE` if the implementation supports setting the depth and stencil resolve modes to different values when one of those modes is `VK_RESOLVE_MODE_NONE`. Otherwise the implementation only supports setting both modes to the same value.
• **independentResolve** is **VK_TRUE** if the implementation supports all combinations of the supported depth and stencil resolve modes, including setting either depth or stencil resolve mode to **VK_RESOLVE_MODE_NONE**. An implementation that supports **independentResolve** **must** also support **independentResolveNone**.

If the **VkPhysicalDeviceDepthStencilResolveProperties** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDepthStencilResolveProperties-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES**

The **VkPhysicalDevicePerformanceQueryPropertiesKHR** structure is defined as:

```c
// Provided by VK_KHR_performance_query
typedef struct VkPhysicalDevicePerformanceQueryPropertiesKHR {
    VkStructureType sType;
    void* pNext;
    VkBool32 allowCommandBufferQueryCopies;
} VkPhysicalDevicePerformanceQueryPropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **allowCommandBufferQueryCopies** is **VK_TRUE** if the performance query pools are allowed to be used with **vkCmdCopyQueryPoolResults**.

If the **VkPhysicalDevicePerformanceQueryPropertiesKHR** structure is included in the **pNext** chain of the **VkPhysicalDeviceProperties2** structure passed to **vkGetPhysicalDeviceProperties2**, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDevicePerformanceQueryPropertiesKHR-sType-sType**
  
  **sType** must be **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR**

The **VkPhysicalDeviceShadingRateImagePropertiesNV** structure is defined as:
// Provided by VK_NV_shading_rate_image

typedef struct VkPhysicalDeviceShadingRateImagePropertiesNV {
    VkStructureType sType;
    void* pNext;
    VkExtent2D shadingRateTexelSize;
    uint32_t shadingRatePaletteSize;
    uint32_t shadingRateMaxCoarseSamples;
} VkPhysicalDeviceShadingRateImagePropertiesNV;

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shadingRateTexelSize` indicates the width and height of the portion of the framebuffer corresponding to each texel in the shading rate image.
- `shadingRatePaletteSize` indicates the maximum number of palette entries supported for the shading rate image.
- `shadingRateMaxCoarseSamples` specifies the maximum number of coverage samples supported in a single fragment. If the product of the fragment size derived from the base shading rate and the number of coverage samples per pixel exceeds this limit, the final shading rate will be adjusted so that its product does not exceed the limit.

If the `VkPhysicalDeviceShadingRateImagePropertiesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These properties are related to the shading rate image feature.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceShadingRateImagePropertiesNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_PROPERTIES_NV`

The `VkPhysicalDeviceTransformFeedbackPropertiesEXT` structure is defined as:
typedef struct VkPhysicalDeviceTransformFeedbackPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxTransformFeedbackStreams;
    uint32_t           maxTransformFeedbackBuffers;
    VkDeviceSize       maxTransformFeedbackBufferSize;
    uint32_t           maxTransformFeedbackStreamDataSize;
    uint32_t           maxTransformFeedbackBufferDataSize;
    uint32_t           maxTransformFeedbackBufferDataStride;
    VkBool32           transformFeedbackQueries;
    VkBool32           transformFeedbackStreamsLinesTriangles;
    VkBool32           transformFeedbackRasterizationStreamSelect;
    VkBool32           transformFeedbackDraw;
} VkPhysicalDeviceTransformFeedbackPropertiesEXT;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- maxTransformFeedbackStreams is the maximum number of vertex streams that can be output from geometry shaders declared with the GeometryStreams capability. If the implementation does not support VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams then maxTransformFeedbackStreams must be set to 1.
- maxTransformFeedbackBuffers is the maximum number of transform feedback buffers that can be bound for capturing shader outputs from the last vertex processing stage.
- maxTransformFeedbackBufferSize is the maximum size that can be specified when binding a buffer for transform feedback in vkCmdBindTransformFeedbackBuffersEXT.
- maxTransformFeedbackStreamDataSize is the maximum amount of data in bytes for each vertex that captured to one or more transform feedback buffers associated with a specific vertex stream.
- maxTransformFeedbackBufferDataSize is the maximum amount of data in bytes for each vertex that can be captured to a specific transform feedback buffer.
- maxTransformFeedbackBufferDataStride is the maximum stride between each capture of vertex data to the buffer.
- transformFeedbackQueries is true if the implementation supports the VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT query type. transformFeedbackQueries is false if queries of this type cannot be created.
- transformFeedbackStreamsLinesTriangles is true if the implementation supports the geometry shader OpExecutionMode of OutputLineStrip and OutputTriangleStrip in addition to OutputPoints when more than one vertex stream is output. If transformFeedbackStreamsLinesTriangles is false the implementation only supports an OpExecutionMode of OutputPoints when more than one vertex stream is output from the geometry shader.
- transformFeedbackRasterizationStreamSelect is true if the implementation supports the GeometryStreams SPIR-V capability and the application can use VkPipelineRasterizationStateStreamCreateInfoEXT to modify which vertex stream output is...
used for rasterization. Otherwise vertex stream \( \texttt{must} \) always be used for rasterization.

- \texttt{transformFeedbackDraw} \texttt{is true if the implementation supports the} \texttt{vkCmdDrawIndirectByteCountEXT} \texttt{function otherwise the function} \texttt{must not be called.}

If the \texttt{VkPhysicalDeviceTransformFeedbackPropertiesEXT} \texttt{structure is included in the} \texttt{pNext chain of the} \texttt{VkPhysicalDeviceProperties2} \texttt{structure passed to} \texttt{vkGetPhysicalDeviceProperties2}, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- \texttt{VUID-VkPhysicalDeviceTransformFeedbackPropertiesEXT-sType-sType} \texttt{sType} \texttt{must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_PROPERTIES_EXT}

The \texttt{VkPhysicalDeviceRayTracingPropertiesNV} structure is defined as:

```c
// Provided by VK_NV_ray_tracing
typedef struct VkPhysicalDeviceRayTracingPropertiesNV {
    VkStructureType  sType;
    void*             pNext;
    uint32_t           shaderGroupHandleSize;
    uint32_t           maxRecursionDepth;
    uint32_t           maxShaderGroupStride;
    uint32_t           shaderGroupBaseAlignment;
    uint64_t           maxGeometryCount;
    uint64_t           maxInstanceCount;
    uint64_t           maxTriangleCount;
    uint32_t           maxDescriptorSetAccelerationStructures;
} VkPhysicalDeviceRayTracingPropertiesNV;
```

- \texttt{sType} \texttt{is the type of this structure.}
- \texttt{pNext} \texttt{is NULL or a pointer to a structure extending this structure.}
- \texttt{shaderGroupHandleSize} \texttt{is the size in bytes of the shader header.}
- \texttt{maxRecursionDepth} \texttt{is the maximum number of levels of recursion allowed in a trace command.}
- \texttt{maxShaderGroupStride} \texttt{is the maximum stride in bytes allowed between shader groups in the shader binding table.}
- \texttt{shaderGroupBaseAlignment} \texttt{is the \texttt{required} alignment in bytes for the base of the shader binding table.}
- \texttt{maxGeometryCount} \texttt{is the maximum number of geometries in the bottom level acceleration structure.}
- \texttt{maxInstanceCount} \texttt{is the maximum number of instances in the top level acceleration structure.}
- \texttt{maxTriangleCount} \texttt{is the maximum number of triangles in all geometries in the bottom level acceleration structure.}
- \texttt{maxDescriptorSetAccelerationStructures} \texttt{is the maximum number of acceleration structure...
descriptors that are allowed in a descriptor set.

If the `VkPhysicalDeviceRayTracingPropertiesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Limits specified by this structure must match those specified with the same name in `VkPhysicalDeviceAccelerationStructurePropertiesKHR` and `VkPhysicalDeviceRayTracingPipelinePropertiesKHR`.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceRayTracingPropertiesNV-sType-sType**
  
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICERAY_TRACING_PROPERTIES_NV`

The `VkPhysicalDeviceAccelerationStructurePropertiesKHR` structure is defined as:

```c
// Provided by VK_KHR_acceleration_structure
typedef struct VkPhysicalDeviceAccelerationStructurePropertiesKHR {
    VkStructureType sType;
    void* pNext;
    uint64_t maxGeometryCount;
    uint64_t maxInstanceCount;
    uint64_t maxPrimitiveCount;
    uint32_t maxPerStageDescriptorAccelerationStructures;
    uint32_t maxPerStageDescriptorUpdateAfterBindAccelerationStructures;
    uint32_t maxDescriptorSetAccelerationStructures;
    uint32_t maxDescriptorSetUpdateAfterBindAccelerationStructures;
    uint32_t minAccelerationStructureScratchOffsetAlignment;
} VkPhysicalDeviceAccelerationStructurePropertiesKHR;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **maxGeometryCount** is the maximum number of geometries in the bottom level acceleration structure.
- **maxInstanceCount** is the maximum number of instances in the top level acceleration structure.
- **maxPrimitiveCount** is the maximum number of triangles or AABBs in all geometries in the bottom level acceleration structure.
- **maxPerStageDescriptorAccelerationStructures** is the maximum number of acceleration structure bindings that can be accessible to a single shader stage in a pipeline layout. Descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR` count against this limit. Only descriptor bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit.
- **maxPerStageDescriptorUpdateAfterBindAccelerationStructures** is similar to `maxPerStageDescriptorAccelerationStructures` but counts descriptor bindings from descriptor
sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `maxDescriptorSetAccelerationStructures` is the maximum number of acceleration structure descriptors that can be included in descriptor bindings in a pipeline layout across all pipeline shader stages and descriptor set numbers. Descriptor bindings with a descriptor type of `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR` count against this limit. Only descriptor bindings in descriptor set layouts created without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set count against this limit.

- `maxDescriptorSetUpdateAfterBindAccelerationStructures` is similar to `maxDescriptorSetAccelerationStructures` but counts descriptor bindings from descriptor sets created with or without the `VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT` bit set.

- `minAccelerationStructureScratchOffsetAlignment` is the minimum required alignment, in bytes, for scratch data passed in to an acceleration structure build command.

If the `VkPhysicalDeviceAccelerationStructurePropertiesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Limits specified by this structure must match those specified with the same name in `VkPhysicalDeviceRayTracingPropertiesNV`.

**Valid Usage (Implicit)**

- `VUID-VkPhysicalDeviceAccelerationStructurePropertiesKHR-sType-sType` sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_PROPERTIES_KHR`

The `VkPhysicalDeviceRayTracingPipelinePropertiesKHR` structure is defined as:

```c
typedef struct VkPhysicalDeviceRayTracingPipelinePropertiesKHR {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           shaderGroupHandleSize;
    uint32_t           maxRayRecursionDepth;
    uint32_t           maxShaderGroupStride;
    uint32_t           shaderGroupBaseAlignment;
    uint32_t           shaderGroupHandleCaptureReplaySize;
    uint32_t           maxRayDispatchInvocationCount;
    uint32_t           shaderGroupHandleAlignment;
    uint32_t           maxRayHitAttributeSize;
} VkPhysicalDeviceRayTracingPipelinePropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `shaderGroupHandleSize` is the size in bytes of the shader header.
• **maxRayRecursionDepth** is the maximum number of levels of ray recursion allowed in a trace command.

• **maxShaderGroupStride** is the maximum stride in bytes allowed between shader groups in the shader binding table.

• **shaderGroupBaseAlignment** is the **required** alignment in bytes for the base of the shader binding table.

• **shaderGroupHandleCaptureReplaySize** is the number of bytes for the information required to do capture and replay for shader group handles.

• **maxRayDispatchInvocationCount** is the maximum number of ray generation shader invocations which **may** be produced by a single `vkCmdTraceRaysIndirectKHR` or `vkCmdTraceRaysKHR` command.

• **shaderGroupHandleAlignment** is the **required** alignment in bytes for each shader binding table entry.

• **maxRayHitAttributeSize** is the maximum size in bytes for a ray attribute structure.

If the `VkPhysicalDeviceRayTracingPipelinePropertiesKHR` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

Limits specified by this structure must match those specified with the same name in `VkPhysicalDeviceRayTracingPropertiesNV`.

---

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceRayTracingPipelinePropertiesKHR-sType-sType
  
  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR`.

The `VkPhysicalDeviceCooperativeMatrixPropertiesNV` structure is defined as:

```
// Provided by VK_NV_cooperative_matrix
typedef struct VkPhysicalDeviceCooperativeMatrixPropertiesNV {
    VkStructureType       sType;
    void*                 pNext;
    VkShaderStageFlags    cooperativeMatrixSupportedStages;
} VkPhysicalDeviceCooperativeMatrixPropertiesNV;
```

- **sType** is the type of this structure.

- **pNext** is `NULL` or a pointer to a structure extending this structure.

- **cooperativeMatrixSupportedStages** is a bitfield of `VkShaderStageFlagBits` describing the shader stages that cooperative matrix instructions are supported in. `cooperativeMatrixSupportedStages` will have the `VK_SHADER_STAGE_COMPUTE_BIT` bit set if any of the physical device's queues support `VK_QUEUE_COMPUTE_BIT`.

If the `VkPhysicalDeviceCooperativeMatrixPropertiesNV` structure is included in the `pNext` chain of the
The `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceCooperativeMatrixPropertiesNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_MATRIX_PROPERTIES_NV`

The `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV` structure is defined as:

```c
// Provided by VK_NV_shader_sm_builtins
typedef struct VkPhysicalDeviceShaderSMBuiltinsPropertiesNV {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           shaderSMCount;
    uint32_t           shaderWarpsPerSM;
} VkPhysicalDeviceShaderSMBuiltinsPropertiesNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `shaderSMCount` is the number of SMs on the device.
- `shaderWarpsPerSM` is the maximum number of simultaneously executing warps on an SM.

If the `VkPhysicalDeviceShaderSMBuiltinsPropertiesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

**Valid Usage (Implicit)**

- VUID-VkPhysicalDeviceShaderSMBuiltinsPropertiesNV-sType-sType
  
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_PROPERTIES_NV`

The `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_texel_buffer_alignment
typedef struct VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkDeviceSize       storageTexelBufferOffsetAlignmentBytes;
    VkBool32           storageTexelBufferOffsetSingleTexelAlignment;
    VkDeviceSize       uniformTexelBufferOffsetAlignmentBytes;
    VkBool32           uniformTexelBufferOffsetSingleTexelAlignment;
} VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT;
```
• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **storageTexelBufferOffsetAlignmentBytes** is a byte alignment that is sufficient for a storage texel buffer of any format.
• **storageTexelBufferOffsetSingleTexelAlignment** indicates whether single texel alignment is sufficient for a storage texel buffer of any format.
• **uniformTexelBufferOffsetAlignmentBytes** is a byte alignment that is sufficient for a uniform texel buffer of any format.
• **uniformTexelBufferOffsetSingleTexelAlignment** indicates whether single texel alignment is sufficient for a uniform texel buffer of any format.

If the VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

If the single texel alignment property is VK_FALSE, then the buffer view’s offset **must** be aligned to the corresponding byte alignment value. If the single texel alignment property is VK_TRUE, then the buffer view’s offset **must** be aligned to the lesser of the corresponding byte alignment value or the size of a singletexel, based on VkBufferViewCreateInfo::format. If the size of a single texel is a multiple of three bytes, then the size of a single component of the format is used instead.

These limits **must** not advertise a larger alignment than the required maximum minimum value of VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment, for any format that supports use as a texel buffer.

### Valid Usage (Implicit)

- VUID-VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT-sType-sType
  
  **sType** **must** be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT

The VkPhysicalDeviceTimelineSemaphoreProperties structure is defined as:

```c
typedef struct VkPhysicalDeviceTimelineSemaphoreProperties {
    VkStructureType sType;
    void* pNext;
    uint64_t maxTimelineSemaphoreValueDifference;
} VkPhysicalDeviceTimelineSemaphoreProperties;
```

or the equivalent

```c
// Provided by VK_KHR_timeline_semaphore
typedef VkPhysicalDeviceTimelineSemaphoreProperties
VkPhysicalDeviceTimelineSemaphorePropertiesKHR;
```
• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure. 

• **maxTimelineSemaphoreValueDifference** indicates the maximum difference allowed by the implementation between the current value of a timeline semaphore and any pending signal or wait operations.

If the `VkPhysicalDeviceTimelineSemaphoreProperties` structure is included in the **pNext** chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceTimelineSemaphoreProperties-sType-sType**

  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES`

The `VkPhysicalDeviceLineRasterizationPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_line_rasterization
typedef struct VkPhysicalDeviceLineRasterizationPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t lineSubPixelPrecisionBits;
} VkPhysicalDeviceLineRasterizationPropertiesEXT;
```

• **sType** is the type of this structure.
• **pNext** is NULL or a pointer to a structure extending this structure.
• **lineSubPixelPrecisionBits** is the number of bits of subpixel precision in framebuffer coordinates \(x_f\) and \(y_f\) when rasterizing line segments.

If the `VkPhysicalDeviceLineRasterizationPropertiesEXT` structure is included in the **pNext** chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceLineRasterizationPropertiesEXT-sType-sType**

  **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT`

The `VkPhysicalDeviceRobustness2PropertiesEXT` structure is defined as:
typedef struct VkPhysicalDeviceRobustness2PropertiesEXT {
    VkStructureType    sType;
    void*              pNext;
    VkDeviceSize       robustStorageBufferAccessSizeAlignment;
    VkDeviceSize       robustUniformBufferAccessSizeAlignment;
} VkPhysicalDeviceRobustness2PropertiesEXT;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.

• **robustStorageBufferAccessSizeAlignment** is the number of bytes that the range of a storage buffer descriptor is rounded up to when used for bounds-checking when robustBufferAccess2 is enabled. This value is either 1 or 4.

• **robustUniformBufferAccessSizeAlignment** is the number of bytes that the range of a uniform buffer descriptor is rounded up to when used for bounds-checking when robustBufferAccess2 is enabled. This value is a power of two in the range [1, 256].

If the VkPhysicalDeviceRobustness2PropertiesEXT structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceRobustness2PropertiesEXT-sType-sType
  
  **sType** must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_PROPERTIES_EXT

The VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV structure is defined as:

typedef struct VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           maxGraphicsShaderGroupCount;
    uint32_t           maxIndirectSequenceCount;
    uint32_t           maxIndirectCommandsTokenCount;
    uint32_t           maxIndirectCommandsStreamCount;
    uint32_t           maxIndirectCommandsTokenOffset;
    uint32_t           maxIndirectCommandsStreamStride;
    uint32_t           minSequencesCountBufferOffsetAlignment;
    uint32_t           minSequencesIndexBufferOffsetAlignment;
    uint32_t           minIndirectCommandsBufferOffsetAlignment;
} VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV;

• **sType** is the type of this structure.

• **pNext** is NULL or a pointer to a structure extending this structure.
• **maxGraphicsShaderGroupCount** is the maximum number of shader groups in `VkGraphicsPipelineShaderGroupsCreateInfoNV`.

• **maxIndirectSequenceCount** is the maximum number of sequences in `VkGeneratedCommandsInfoNV` and in `VkGeneratedCommandsMemoryRequirementsInfoNV`.

• **maxIndirectCommandsLayoutTokenCount** is the maximum number of tokens in `VkIndirectCommandsLayoutCreateInfoNV`.

• **maxIndirectCommandsStreamCount** is the maximum number of streams in `VkIndirectCommandsLayoutCreateInfoNV`.

• **maxIndirectCommandsTokenOffset** is the maximum offset in `VkIndirectCommandsLayoutTokenNV`.

• **maxIndirectCommandsStreamStride** is the maximum stream stride in `VkIndirectCommandsLayoutCreateInfoNV`.

• **minSequenceCountBufferOffsetAlignment** is the minimum alignment for memory addresses optionally used in `VkGeneratedCommandsInfoNV`.

• **minSequenceIndexBufferOffsetAlignment** is the minimum alignment for memory addresses optionally used in `VkGeneratedCommandsInfoNV`.

• **minIndirectCommandsBufferOffsetAlignment** is the minimum alignment for memory addresses used in `VkIndirectCommandsStreamNV` and as preprocess buffer in `VkGeneratedCommandsInfoNV`.

If the `VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGet PhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV-sType**

  * `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_PROPERTIES_NV`.

The `VkPhysicalDevicePortabilitySubsetPropertiesKHR` structure is defined as:

```c
typedef struct VkPhysicalDevicePortabilitySubsetPropertiesKHR {
    VkStructureType sType;
    void* pNext;
    uint32_t minVertexInputBindingStrideAlignment;
} VkPhysicalDevicePortabilitySubsetPropertiesKHR;
```

- **sType** is the type of this structure.

- **pNext** is `NULL` or a pointer to a structure extending this structure.

- **minVertexInputBindingStrideAlignment** indicates the minimum alignment for vertex input strides. `VkVertexInputBindingDescription::stride` must be a multiple of, and at least as large as, this value.
If the \texttt{VkPhysicalDevicePortabilitySubsetPropertiesKHR} structure is included in the \texttt{pNext} chain of the \texttt{VkPhysicalDeviceProperties2} structure passed to \texttt{vkGetPhysicalDeviceProperties2}, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PHYSICAL DEVICE_PORTABILITY_SUBSET_PROPERTIES_KHR}.

The \texttt{VkPhysicalDeviceFragmentShadingRatePropertiesKHR} structure is defined as:

```c
typedef struct VkPhysicalDeviceFragmentShadingRatePropertiesKHR {
    VkStructureType          sType;
    void*                    pNext;
    VkExtent2D               minFragmentShadingRateAttachmentTexelSize;
    VkExtent2D               maxFragmentShadingRateAttachmentTexelSize;
    uint32_t                 maxFragmentShadingRateAttachmentTexelSizeAspectRatio;
    VkBool32                 primitiveFragmentShadingRateWithMultipleViewports;
    VkBool32                 layeredShadingRateAttachments;
    VkBool32                 fragmentShadingRateNonTrivialCombinerOps;
    VkExtent2D               maxFragmentSize;
    uint32_t                 maxFragmentSizeAspectRatio;
    uint32_t                 maxFragmentShadingRateCoverageSamples;
    VkSampleCountFlagBits    maxFragmentShadingRateRasterizationSamples;
    VkBool32                 fragmentShadingRateWithShaderDepthStencilWrites;
    VkBool32                 fragmentShadingRateWithSampleMask;
    VkBool32                 fragmentShadingRateWithShaderSampleMask;
    VkBool32                 fragmentShadingRateWithConservativeRasterization;
    VkBool32                 fragmentShadingRateWithFragmentShaderInterlock;
    VkBool32                 fragmentShadingRateWithCustomSampleLocations;
    VkBool32                 fragmentShadingRateStrictMultiplyCombiner;
} VkPhysicalDeviceFragmentShadingRatePropertiesKHR;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is NULL or a pointer to a structure extending this structure.
- \texttt{minFragmentShadingRateAttachmentTexelSize} indicates minimum supported width and height of the portion of the framebuffer corresponding to each texel in a fragment shading rate attachment. Each value must be less than or equal to the values in \texttt{maxFragmentShadingRateAttachmentTexelSize}. Each value must be a power-of-two. It must be (0,0) if the \texttt{attachmentFragmentShadingRate} feature is not supported.
- \texttt{maxFragmentShadingRateAttachmentTexelSize} indicates maximum supported width and height of the portion of the framebuffer corresponding to each texel in a fragment shading rate attachment. Each value must be greater than or equal to the values in \texttt{minFragmentShadingRateAttachmentTexelSize}. Each value must be a power-of-two. It must be (0,0) if the \texttt{attachmentFragmentShadingRate} feature is not supported.
• \texttt{maxFragmentShadingRateAttachmentTexelSizeAspectRatio} indicates the maximum ratio between the width and height of the portion of the framebuffer corresponding to each texel in a fragment shading rate attachment. \texttt{maxFragmentShadingRateAttachmentTexelSizeAspectRatio} must be a power-of-two value, and must be less than or equal to \( \frac{\text{maxFragmentShadingRateAttachmentTexelSize.width}}{\text{minFragmentShadingRateAttachmentTexelSize.height}} \) / \( \frac{\text{minFragmentShadingRateAttachmentTexelSize.width}}{\text{maxFragmentShadingRateAttachmentTexelSize.height}} \). It must be 0 if the \texttt{attachmentFragmentShadingRate} feature is not supported.

• \texttt{primitiveFragmentShadingRateWithMultipleViewports} specifies whether the \texttt{primitive fragment shading rate} can be used when multiple viewports are used. If this value is \texttt{VK_FALSE}, only a single viewport must be used, and applications must not write to the \texttt{ViewportMaskNV} or \texttt{ViewportIndex} built-in when setting \texttt{PrimitiveShadingRateKHR}. It must be \texttt{VK_FALSE} if the \texttt{VK_EXT_shader_viewport_index_layer} extension, or the \texttt{geometryShader} feature is not supported, or if the \texttt{primitiveFragmentShadingRate} feature is not supported.

• \texttt{layeredShadingRateAttachments} specifies whether a shading rate attachment image view can be created with multiple layers. If this value is \texttt{VK_FALSE}, when creating an image view with a usage that includes \texttt{VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR}, \texttt{layerCount} must be 1. It must be \texttt{VK_FALSE} if the multiview feature, the \texttt{VK_EXT_shader_viewport_index_layer} extension, or the \texttt{geometryShader} feature is not supported, or if the \texttt{attachmentFragmentShadingRate} feature is not supported.

• \texttt{fragmentShadingRateNonTrivialCombinerOps} specifies whether \texttt{VkFragmentShadingRateCombinerOpKHR} enums other than \texttt{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_KEEP_KHR} or \texttt{VK_FRAGMENT_SHADING_RATE_COMBINER_OP_REPLACE_KHR} can be used. It must be \texttt{VK_FALSE} unless either the \texttt{primitiveFragmentShadingRate} or \texttt{attachmentFragmentShadingRate} feature is supported.

• \texttt{maxFragmentSize} indicates the maximum supported width and height of a fragment. Its width and height members must both be power-of-two values. This limit is purely informational, and is not validated.

• \texttt{maxFragmentSizeAspectRatio} indicates the maximum ratio between the width and height of a fragment. \texttt{maxFragmentSizeAspectRatio} must be a power-of-two value, and must be less than or equal to the maximum of the width and height members of \texttt{maxFragmentSize}. This limit is purely informational, and is not validated.

• \texttt{maxFragmentShadingRateCoverageSamples} specifies the maximum number of coverage samples supported in a single fragment. \texttt{maxFragmentShadingRateCoverageSamples} must be less than or equal to the product of the width and height members of \texttt{maxFragmentSize}, and the sample count reported by \texttt{maxFragmentShadingRateRasterizationSamples}. \texttt{maxFragmentShadingRateCoverageSamples} must be less than or equal to \( \text{maxSampleMaskWords} \times 32 \) if \texttt{fragmentShadingRateWithShaderSampleMask} is supported. This limit is purely informational, and is not validated.

• \texttt{maxFragmentShadingRateRasterizationSamples} specifies the maximum sample rate supported when a fragment covers multiple pixels. This limit is purely informational, and is not validated.

• \texttt{fragmentShadingRateWithShaderDepthStencilWrites} specifies whether the implementation supports writing \texttt{FragDepth} or \texttt{FragStencilRefEXT} from a fragment shader for multi-pixel fragments. If this value is \texttt{VK_FALSE}, writing to those built-ins will clamp the fragment shading
rate to (1,1).

- fragmentShadingRateWithSampleMask specifies whether the implementation supports setting valid bits of VkPipelineMultisampleStateCreateInfo::pSampleMask to 0 for multi-pixel fragments. If this value is VK_FALSE, zeroing valid bits in the sample mask will clamp the fragment shading rate to (1,1).

- fragmentShadingRateWithShaderSampleMask specifies whether the implementation supports reading or writing SampleMask for multi-pixel fragments. If this value is VK_FALSE, using that built-in will clamp the fragment shading rate to (1,1).

- fragmentShadingRateWithConservativeRasterization specifies whether conservative rasterization is supported for multi-pixel fragments. It must be VK_FALSE if VK_EXT_conservative_rasterization is not supported. If this value is VK_FALSE, using conservative rasterization will clamp the fragment shading rate to (1,1).

- fragmentShadingRateWithFragmentShaderInterlock specifies whether fragment shader interlock is supported for multi-pixel fragments. It must be VK_FALSE if VK_EXT_fragment_shader_interlock is not supported. If this value is VK_FALSE, using fragment shader interlock will clamp the fragment shading rate to (1,1).

- fragmentShadingRateWithCustomSampleLocations specifies whether custom sample locations are supported for multi-pixel fragments. It must be VK_FALSE if VK_EXT_sample_locations is not supported. If this value is VK_FALSE, using custom sample locations will clamp the fragment shading rate to (1,1).

- fragmentShadingRateStrictMultiplyCombiner specifies whether VK_FRAGMENT_SHADING_RATE_COMBINER_OP_MUL_KHR accurately performs a multiplication or not. Implementations where this value is VK_FALSE will instead combine rates with an addition. If fragmentShadingRateNonTrivialCombinerOps is VK_FALSE, implementations must report this as VK_FALSE. If fragmentShadingRateNonTrivialCombinerOps is VK_TRUE, implementations should report this as VK_TRUE.

Note

Multiplication of the combiner rates using the fragment width/height in linear space is equivalent to an addition of those values in log2 space. Some implementations inadvertently implemented an addition in linear space due to unclear requirements originating outside of this specification. This resulted in fragmentShadingRateStrictMultiplyCombiner being added. Fortunately, this only affects situations where a rate of 1 in either dimension is combined with another rate of 1. All other combinations result in the exact same result as if multiplication was performed in linear space due to the clamping logic, and the fact that both the sum and product of 2 and 2 are equal. In many cases, this limit will not affect the correct operation of applications.

If the VkPhysicalDeviceFragmentShadingRatePropertiesKHR structure is included in the pNext chain of the VkPhysicalDeviceProperties2 structure passed to vkGetPhysicalDeviceProperties2, it is filled in with each corresponding implementation-dependent property.

These properties are related to fragment shading rates.
The `VkPhysicalDeviceFragmentShadingRateEnumsPropertiesNV` structure is defined as:

```c
// Provided by VK_NV_fragment_shading_rate Enums
typedef struct VkPhysicalDeviceFragmentShadingRateEnumsPropertiesNV {
    VkStructureType sType;
    void* pNext;
    VkSampleCountFlagBits maxFragmentShadingRateInvocationCount;
} VkPhysicalDeviceFragmentShadingRateEnumsPropertiesNV;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxFragmentShadingRateInvocationCount` indicates the maximum number of fragment shader invocations per fragment supported in pipeline, primitive, and attachment fragment shading rates.

If the `VkPhysicalDeviceFragmentShadingRateEnumsPropertiesNV` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

These properties are related to fragment shading rates.

The `VkPhysicalDeviceCustomBorderColorPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_custom_border_color
typedef struct VkPhysicalDeviceCustomBorderColorPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    uint32_t maxCustomBorderColorSamplers;
} VkPhysicalDeviceCustomBorderColorPropertiesEXT;
```
• `maxCustomBorderColorSamplers` indicates the maximum number of samplers with custom border colors which can simultaneously exist on a device.

If the `VkPhysicalDeviceCustomBorderColorPropertiesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceCustomBorderColorPropertiesEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_PROPERTIES_EXT`  

The `VkPhysicalDeviceProvokingVertexPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_provoking_vertex
typedef struct VkPhysicalDeviceProvokingVertexPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 provokingVertexModePerPipeline;
    VkBool32 transformFeedbackPreservesTriangleFanProvokingVertex;
} VkPhysicalDeviceProvokingVertexPropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `provokingVertexModePerPipeline` indicates whether the implementation supports graphics pipelines with different provoking vertex modes within the same renderpass instance.
- `transformFeedbackPreservesTriangleFanProvokingVertex` indicates whether the implementation can preserve the provoking vertex order when writing triangle fan vertices to transform feedback.

If the `VkPhysicalDeviceProvokingVertexPropertiesEXT` structure is included in the `pNext` chain of the `VkPhysicalDeviceProperties2` structure passed to `vkGetPhysicalDeviceProperties2`, it is filled in with each corresponding implementation-dependent property.

### Valid Usage (Implicit)

- `VUID-VkPhysicalDeviceProvokingVertexPropertiesEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROVOKING_VERTEX_PROPERTIES_EXT`  

### 42.1. Limit Requirements

The following table specifies the **required** minimum/maximum for all Vulkan graphics implementations. Where a limit corresponds to a fine-grained device feature which is **optional**, the feature name is listed with two **required** limits, one when the feature is supported and one when it is not.

---

2240 | Chapter 42. Limits
is not supported. If an implementation supports a feature, the limits reported are the same whether or not the feature is enabled.

**Table 53. Required Limit Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Limit</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t</td>
<td>maxImageDimension1D</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxImageDimension2D</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxImageDimension3D</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxImageDimensionCube</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxImageArrayLayers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTexelBufferElements</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxUniformBufferRange</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxStorageBufferRange</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPushConstantsSize</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxMemoryAllocationCount</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSamplerAllocationCount</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>bufferImageGranularity</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>sparseAddressSpaceSize</td>
<td>sparseBinding</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxBoundDescriptorSets</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorSamplers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUniformBuffers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorStorageBuffers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorSampledImages</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorStorageImages</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorInputAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageResources</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSamplers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageBuffers</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSampledImages</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetInputAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputAttributes</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputBindings</td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputAttributeOffset</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexInputBindingStride</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxVertexOutputComponents</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationGenerationLevel</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationPatchSize</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlPerVertexInputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlPerVertexOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlPerPatchOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationControlTotalOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationEvaluationInputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTessellationEvaluationOutputComponents</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryShaderInvocations</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryInputComponents</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryOutputComponents</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryOutputVertices</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryTotalOutputComponents</td>
<td>geometryShader</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentInputComponents</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentOutputAttachments</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentDualSrcAttachments</td>
<td>dualSrcBlend</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFragmentCombinedOutputResources</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxComputeSharedMemorySize</td>
<td>-</td>
</tr>
<tr>
<td>3 × uint32_t</td>
<td>maxComputeWorkGroupCount</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxComputeWorkGroupInvocations</td>
<td>-</td>
</tr>
<tr>
<td>3 × uint32_t</td>
<td>maxComputeWorkGroupSize</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>subPixelPrecisionBits</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>subTexelPrecisionBits</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>mipmapPrecisionBits</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDrawIndexedIndexValue</td>
<td>fullDrawIndexedUint32</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDrawIndirectCount</td>
<td>multiDrawIndirect</td>
</tr>
<tr>
<td>float</td>
<td>maxSamplerLodBias</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>maxSamplerAnisotropy</td>
<td>samplerAnisotropy</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxViewports</td>
<td>multiViewport</td>
</tr>
<tr>
<td>2 × uint32_t</td>
<td>maxViewportDimensions</td>
<td>-</td>
</tr>
<tr>
<td>2 × float</td>
<td>viewportBoundsRange</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>viewportSubPixelBits</td>
<td>-</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>size_t</td>
<td>minMemoryMapAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>minTexelBufferOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>minUniformBufferOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>minStorageBufferOffsetAlignment</td>
<td>-</td>
</tr>
<tr>
<td>int32_t</td>
<td>minTexelOffset</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTexelOffset</td>
<td>-</td>
</tr>
<tr>
<td>int32_t</td>
<td>minTexelGatherOffset</td>
<td>shaderImageGatherExtended</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTexelGatherOffset</td>
<td>shaderImageGatherExtended</td>
</tr>
<tr>
<td>float</td>
<td>minInterpolationOffset</td>
<td>sampleRateShading</td>
</tr>
<tr>
<td>float</td>
<td>maxInterpolationOffset</td>
<td>sampleRateShading</td>
</tr>
<tr>
<td>uint32_t</td>
<td>subPixelInterpolationOffsetBits</td>
<td>sampleRateShading</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferWidth</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferHeight</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxFramebufferLayers</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferColorSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferDepthSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferStencilSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>framebufferNoAttachmentsSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxColorAttachments</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageColorSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageIntegerSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageDepthSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>sampledImageStencilSampleCounts</td>
<td>-</td>
</tr>
<tr>
<td>VkSampleCountFlags</td>
<td>storageImageSampleCounts</td>
<td>shaderStorageImageMultisample</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSampleMaskWords</td>
<td>-</td>
</tr>
<tr>
<td>VkBool32</td>
<td>timestampComputeAndGraphics</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>timestampPeriod</td>
<td>-</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxClipDistances</td>
<td>shaderClipDistance</td>
</tr>
</tbody>
</table>

Chapter 42. Limits | 2243
<table>
<thead>
<tr>
<th>Type</th>
<th>Limit</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t</td>
<td>maxCullDistances</td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxCombinedClipAndCullDistances</td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>discreteQueuePriorities</td>
<td></td>
</tr>
<tr>
<td>2 × float</td>
<td>pointSizeRange</td>
<td>largePoints</td>
</tr>
<tr>
<td>2 × float</td>
<td>lineWidthRange</td>
<td>wideLines</td>
</tr>
<tr>
<td>float</td>
<td>pointSizeGranularity</td>
<td>largePoints</td>
</tr>
<tr>
<td>float</td>
<td>lineWidthGranularity</td>
<td>wideLines</td>
</tr>
<tr>
<td>VkBool32</td>
<td>strictLines</td>
<td></td>
</tr>
<tr>
<td>VkBool32</td>
<td>standardSampleLocations</td>
<td></td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>optimalBufferCopyOffsetAlignment</td>
<td></td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>optimalBufferCopyRowPitchAlignment</td>
<td></td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>nonCoherentAtomSize</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDiscardRectangles</td>
<td>VK_EXT_discard_rectangles</td>
</tr>
<tr>
<td>VkBool32</td>
<td>filterMinmaxSingleComponentFormats</td>
<td>VK_EXT_sampler_filter_minmax</td>
</tr>
<tr>
<td>VkBool32</td>
<td>filterMinmaxImageComponentMapping</td>
<td>VK_EXT_sampler_filter_minmax</td>
</tr>
<tr>
<td>float</td>
<td>primitiveOverestimationSize</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>maxExtraPrimitiveOverestimationSize</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>float</td>
<td>extraPrimitiveOverestimationSizeGranularity</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>degenerateTriangleRasterized</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>float</td>
<td>degenerateLinesRasterized</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fullyCoveredFragmentShaderInputVariable</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>VkBool32</td>
<td>conservativeRasterizationPostDepthCoverage</td>
<td>VK_EXT_conservative_rasterization</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxUpdateAfterBindDescriptorsInAllPools</td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderUniformBufferArrayNonUniformIndexingNative</td>
<td></td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderSampledImageArrayNonUniformIndexingNative</td>
<td></td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderStorageBufferArrayNonUniformIndexingNative</td>
<td></td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderStorageImageArrayNonUniformIndexingNative</td>
<td></td>
</tr>
<tr>
<td>VkBool32</td>
<td>shaderInputAttachmentArrayNonUniformIndexingNative</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindSamplers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindUniformBuffers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindStorageBuffers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindSampledImages</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindStorageImages</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageDescriptorUpdateAfterBindInputAttachments</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxPerStageUpdateAfterBindResources</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindSamplers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindUniformBuffers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindStorageBuffers</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindSampledImages</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindStorageImages</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDescriptorSetUpdateAfterBindInputAttachments</code></td>
<td>descriptorIndexing</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxVertexAttribDivisor</code></td>
<td><code>VK_EXT_vertex_attribute_divisor</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxDrawMeshTasksCount</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTaskWorkGroupInvocations</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTaskWorkGroupSize</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTaskTotalMemorySize</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTaskOutputCount</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMeshWorkGroupInvocations</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMeshWorkGroupSize</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMeshTotalMemorySize</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMeshOutputVertices</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMeshOutputPrimitives</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxMeshMultiviewViewCount</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>meshOutputPerVertexGranularity</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>meshOutputPerPrimitiveGranularity</code></td>
<td><code>VK_NV_mesh_shader</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTransformFeedbackStreams</code></td>
<td><code>VK_EXT_transform_feedback</code></td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxTransformFeedbackBuffers</code></td>
<td><code>VK_EXT_transform_feedback</code></td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td><code>maxTransformFeedbackBufferSize</code></td>
<td><code>VK_EXT_transform_feedback</code></td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTransformFeedbackStreamDataSize</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTransformFeedbackBufferDataSize</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTransformFeedbackBufferDataStride</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>VkBool32</td>
<td>transformFeedbackQueries</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>VkBool32</td>
<td>transformFeedbackStreamsLinesTriangles</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>VkBool32</td>
<td>transformFeedbackRasterizationStreamSelect</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>VkBool32</td>
<td>transformFeedbackDraw</td>
<td>VK_EXT_transform_feedback</td>
</tr>
<tr>
<td>VkExtent2D</td>
<td>minFragmentDensityTexelSize</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>VkExtent2D</td>
<td>maxFragmentDensityTexelSize</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>VkBool32</td>
<td>fragmentDensityInvocations</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>VkBool32</td>
<td>subsampledLoads</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>VkBool32</td>
<td>subsampledCoarseReconstructionEarlyAccess</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxSubsampledArrayLayers</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetSubsampledSamplers</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxGeometryCount</td>
<td>VK_NV_ray_tracing, VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxInstanceCount</td>
<td>VK_NV_ray_tracing, VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>shaderGroupHandleSize</td>
<td>VK_NV_ray_tracing, VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxShaderGroupStride</td>
<td>VK_NV_ray_tracing, VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>shaderGroupBaseAlignment</td>
<td>VK_NV_ray_tracing, VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxRecursionDepth</td>
<td>VK_NV_ray_tracing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxTriangleCount</td>
<td>VK_NV_ray_tracing</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorAccelerationStructures</td>
<td>VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxPerStageDescriptorUpdateAfterBindAccelerationStructures</td>
<td>VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetAccelerationStructures</td>
<td>VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>maxDescriptorSetUpdateAfterBindAccelerationStructures</td>
<td>VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>uint32_t</td>
<td>minAccelerationStructureScratchOffsetAlignment</td>
<td>VK_KHR_acceleration_structure</td>
</tr>
<tr>
<td>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxRayRecursionDepth</td>
<td>VK_KHR_ray_tracing_pipeline</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>shaderGroupHandleCaptureReplaySize</td>
<td>VK_KHR_ray_tracing_pipeline</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxRayDispatchInvocationCount</td>
<td>VK_KHR_ray_tracing_pipeline</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>shaderGroupHandleAlignment</td>
<td>VK_KHR_ray_tracing_pipeline</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxRayHitAttributeSize</td>
<td>VK_KHR_ray_tracing_pipeline</td>
</tr>
<tr>
<td><code>uint64_t</code></td>
<td>maxTimelineSemaphoreValueDifference</td>
<td>timelineSemaphore</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>lineSubPixelPrecisionBits</td>
<td>VK_EXT_line_rasterization</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxCustomBorderColorSamplers</td>
<td>VK_EXT_custom_border_color</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td>robustStorageBufferAccessSizeAlignment</td>
<td>VK_EXT_robustness2</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td>robustUniformBufferAccessSizeAlignment</td>
<td>VK_EXT_robustness2</td>
</tr>
<tr>
<td><code>2 × uint32_t</code></td>
<td>minFragmentShadingRateAttachmentTexelSize</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td><code>2 × uint32_t</code></td>
<td>maxFragmentShadingRateAttachmentTexelSize</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxFragmentShadingRateAttachmentTexelSizeAspectRatio</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>primitiveFragmentShadingRateWithMultipleViewports</td>
<td>primitiveFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>layeredShadingRateAttachments</td>
<td>attachmentFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateNonTrivialCombinerOps</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>2 × uint32_t</code></td>
<td>maxFragmentSize</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxFragmentSizeAspectRatio</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td>maxFragmentShadingRateCoverageSamples</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkSampleCountFlagBits</code></td>
<td>maxFragmentShadingRateRasterizationSamples</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateWithShaderDepthStencilWrite</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateWithSampleMask</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateWithShaderSampleMask</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateWithConservativeRasterization</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateWithFragmentShaderInterlock</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateWithCustomSampleLocations</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td>fragmentShadingRateStrictMultiplyCombiner</td>
<td>pipelineFragmentShadingRate</td>
</tr>
<tr>
<td><code>VkSampleCountFlagBits</code></td>
<td>maxFragmentShadingRateInvocationCount</td>
<td>supersampleFragmentShadingRates</td>
</tr>
</tbody>
</table>

*Table 54. Required Limits*
<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupported Limit</th>
<th>Supported Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxImageDimension1D</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension2D</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimension3D</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxImageDimensionCube</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxImageArrayLayers</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxTexelBufferElements</td>
<td>-</td>
<td>65536</td>
<td>min</td>
</tr>
<tr>
<td>maxUniformBufferRange</td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td>maxStorageBufferRange</td>
<td>-</td>
<td>$2^{27}$</td>
<td>min</td>
</tr>
<tr>
<td>maxPushConstantsSize</td>
<td>-</td>
<td>128</td>
<td>min</td>
</tr>
<tr>
<td>maxMemoryAllocationCount</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerAllocationCount</td>
<td>-</td>
<td>4000</td>
<td>min</td>
</tr>
<tr>
<td>bufferImageGranularity</td>
<td>-</td>
<td>131072</td>
<td>max</td>
</tr>
<tr>
<td>sparseAddressSpaceSize</td>
<td>0</td>
<td>$2^{31}$</td>
<td>min</td>
</tr>
<tr>
<td>maxBoundDescriptorSets</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorSamplers</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers</td>
<td>-</td>
<td>12</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorSampledImages</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorInputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageResources</td>
<td>-</td>
<td>128$^2$</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSamplers</td>
<td>-</td>
<td>96$^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>-</td>
<td>72$^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>-</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers</td>
<td>-</td>
<td>24$^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages</td>
<td>-</td>
<td>96$^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
<td>24$^8$</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>maxVertexInputAttributes</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputBindings</td>
<td>-</td>
<td>16(^{10})</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputAttributeOffset</td>
<td>-</td>
<td>2047</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputBindingStride</td>
<td>-</td>
<td>2048</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexOutputComponents</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationGenerationLevel</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationPatchSize</td>
<td>0</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlPerVertexInputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlPerVertexOutputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlPerPatchOutputComponents</td>
<td>0</td>
<td>120</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationControlTotalOutputComponents</td>
<td>0</td>
<td>2048</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationEvaluationInputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxTessellationEvaluationOutputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryShaderInvocations</td>
<td>0</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryInputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryOutputComponents</td>
<td>0</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryOutputVertices</td>
<td>0</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxGeometryTotalOutputComponents</td>
<td>0</td>
<td>1024</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentInputComponents</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentOutputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentDualSrcAttachments</td>
<td>0</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentCombinedOutputResources</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeSharedMemorySize</td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupCount</td>
<td>-</td>
<td>(65535,65535,65535)</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupInvocations</td>
<td>-</td>
<td>128</td>
<td>min</td>
</tr>
<tr>
<td>maxComputeWorkGroupSize</td>
<td>-</td>
<td>(128,128,64)</td>
<td>min</td>
</tr>
<tr>
<td>subPixelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>subTexelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>mipmapPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawIndexedIndexValue</td>
<td>(2^{24}-1)</td>
<td>(2^{32}-1)</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawIndirectCount</td>
<td>1</td>
<td>(2^{16}-1)</td>
<td>min</td>
</tr>
<tr>
<td>maxSamplerLodBias</td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>---------------------</td>
<td>------------</td>
</tr>
<tr>
<td>maxSamplerAnisotropy</td>
<td>1</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxViewports</td>
<td>1</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxViewportDimensions</td>
<td>-</td>
<td>(4096,4096)³</td>
<td>min</td>
</tr>
<tr>
<td>viewportBoundsRange</td>
<td>-</td>
<td>(-8192,8191)⁴</td>
<td>(max,min)</td>
</tr>
<tr>
<td>viewportSubPixelBits</td>
<td>-</td>
<td>0</td>
<td>min</td>
</tr>
<tr>
<td>minMemoryMapAlignment</td>
<td>-</td>
<td>64</td>
<td>min</td>
</tr>
<tr>
<td>minTexelBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minUniformBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minStorageBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minTexelOffset</td>
<td>-</td>
<td>-8</td>
<td>max</td>
</tr>
<tr>
<td>maxTexelOffset</td>
<td>-</td>
<td>7</td>
<td>min</td>
</tr>
<tr>
<td>minTexelGatherOffset</td>
<td>0</td>
<td>-8</td>
<td>max</td>
</tr>
<tr>
<td>maxTexelGatherOffset</td>
<td>0</td>
<td>7</td>
<td>min</td>
</tr>
<tr>
<td>minInterpolationOffset</td>
<td>0.0</td>
<td>-0.5⁵</td>
<td>max</td>
</tr>
<tr>
<td>maxInterpolationOffset</td>
<td>0.0</td>
<td>0.5 - (1 ULP)⁵</td>
<td>min</td>
</tr>
<tr>
<td>subPixelInterpolationOffsetBits</td>
<td>0</td>
<td>4⁵</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferWidth</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferHeight</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>maxFramebufferLayers</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>framebufferColorSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
<td></td>
</tr>
<tr>
<td>framebufferDepthSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
<td></td>
</tr>
<tr>
<td>framebufferStencilSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
<td></td>
</tr>
<tr>
<td>framebufferNoAttachmentsSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
<td></td>
</tr>
<tr>
<td>maxColorAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>sampledImageColorSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>sampledImageIntegerSampleCounts</code></td>
<td>-</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td><code>sampledImageDepthSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>sampledImageStencilSampleCounts</code></td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>storageImageSampleCounts</code></td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td><code>maxSampleMaskWords</code></td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td><code>timestampComputeAndGraphics</code></td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td><code>timestampPeriod</code></td>
<td>-</td>
<td>-</td>
<td>duration</td>
</tr>
<tr>
<td><code>maxClipDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxCullDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>maxCombinedClipAndCullDistances</code></td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td><code>discreteQueuePriorities</code></td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td><code>pointSizeRange</code></td>
<td>(1.0,1.0)</td>
<td>(1.0,64.0 - ULP)</td>
<td>(max,min)</td>
</tr>
<tr>
<td><code>lineWidthRange</code></td>
<td>(1.0,1.0)</td>
<td>(1.0,8.0 - ULP)</td>
<td>(max,min)</td>
</tr>
<tr>
<td><code>pointSizeGranularity</code></td>
<td>0.0</td>
<td>1.0</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td><code>lineWidthGranularity</code></td>
<td>0.0</td>
<td>1.0</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td><code>strictLines</code></td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td><code>standardSampleLocations</code></td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>optimalBufferCopyOffsetAlignment</td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td>optimalBufferCopyRowPitchAlignment</td>
<td>-</td>
<td>-</td>
<td>recommendation</td>
</tr>
<tr>
<td>nonCoherentAtomSize</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>maxPushDescriptors</td>
<td>-</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxMultiviewViewCount</td>
<td>-</td>
<td>6</td>
<td>min</td>
</tr>
<tr>
<td>maxMultiviewInstanceIndex</td>
<td>-</td>
<td>$2^{27} - 1$</td>
<td>min</td>
</tr>
<tr>
<td>maxDiscardRectangles</td>
<td>0</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>sampleLocationSampleCounts</td>
<td>-</td>
<td>VK_SAMPLE_COUNT_4_BIT</td>
<td>min</td>
</tr>
<tr>
<td>maxSampleLocationGridSize</td>
<td>-</td>
<td>(1,1)</td>
<td>min</td>
</tr>
<tr>
<td>sampleLocationCoordinateRange</td>
<td>-</td>
<td>(0.0, 0.9375)</td>
<td>(max, min)</td>
</tr>
<tr>
<td>sampleLocationSubPixelBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>variableSampleLocations</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>minImportedHostPointerAlignment</td>
<td>-</td>
<td>65536</td>
<td>max</td>
</tr>
<tr>
<td>perViewPositionAllComponents</td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>filterMinmaxSingleComponentFormats</td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>filterMinmaxImageComponentMapping</td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>advancedBlendMaxColorAttachments</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>advancedBlendIndependentBlend</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>advancedBlendNonPremultipliedSrcColor</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>advancedBlendNonPremultipliedDstColor</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>advancedBlendCorrelatedOverlap</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>advancedBlendAllOperations</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>maxPerSetDescriptors</td>
<td>-</td>
<td>1024</td>
<td>min</td>
</tr>
<tr>
<td>maxMemoryAllocationSize</td>
<td>-</td>
<td>$2^{30}$</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>primitiveOverestimationSize</td>
<td>-</td>
<td>0.0</td>
<td>min</td>
</tr>
<tr>
<td>maxExtraPrimitiveOverestimationSize</td>
<td>-</td>
<td>0.0</td>
<td>min</td>
</tr>
<tr>
<td>extraPrimitiveOverestimationSizeGranularity</td>
<td>-</td>
<td>0.0</td>
<td>min</td>
</tr>
<tr>
<td>primitiveUnderestimation</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>conservativePointAndLineRasterization</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>degenerateTrianglesRasterized</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>degenerateLinesRasterized</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>fullyCoveredFragmentShaderInputVariable</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>conservativeRasterizationPostDepthCoverage</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>maxUpdateAfterBindDescriptorsInAllPools</td>
<td>0</td>
<td>500000</td>
<td>min</td>
</tr>
<tr>
<td>shaderUniformBufferArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>shaderSampledImageArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>shaderStorageBufferArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>shaderStorageImageArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>shaderInputAttachmentArrayNonUniformIndexingNative</td>
<td>-</td>
<td>false</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindSamplers</td>
<td>$0^9$</td>
<td>$500000^9$</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindUniformBuffers</td>
<td>$0^9$</td>
<td>$12^9$</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindStorageBuffers</td>
<td>$0^9$</td>
<td>$500000^9$</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindSampledImages</td>
<td>$0^9$</td>
<td>$500000^9$</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindStorageImages</td>
<td>$0^9$</td>
<td>$500000^9$</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInputAttachments</td>
<td>$0^9$</td>
<td>$4^9$</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageUpdateAfterBindResources</td>
<td>$0^9$</td>
<td>$500000^9$</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSamplers</td>
<td>0⁹</td>
<td>500000⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffers</td>
<td>0⁹</td>
<td>72⁸⁹</td>
<td>min, $n \times$ PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindUniformBuffersDynamic</td>
<td>0⁹</td>
<td>8⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageBuffers</td>
<td>0⁹</td>
<td>500000⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageBuffersDynamic</td>
<td>0⁹</td>
<td>4⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindSampledImages</td>
<td>0⁹</td>
<td>500000⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindStorageImages</td>
<td>0⁹</td>
<td>500000⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindInputAttachments</td>
<td>0⁹</td>
<td>4⁹</td>
<td>min</td>
</tr>
<tr>
<td>maxInlineUniformBlockSize</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindInlineUniformBlocks</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexAttribDivisor</td>
<td>-</td>
<td>$2^{16}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxDrawMeshTasksCount</td>
<td>-</td>
<td>$2^{16}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxTaskWorkGroupInvocations</td>
<td>-</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxTaskWorkGroupSize</td>
<td>-</td>
<td>(32,1,1)</td>
<td>min</td>
</tr>
<tr>
<td>maxTaskTotalMemorySize</td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td>maxTaskOutputCount</td>
<td>-</td>
<td>$2^{16}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxMeshWorkGroupInvocations</td>
<td>-</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxMeshWorkGroupSize</td>
<td>-</td>
<td>(32,1,1)</td>
<td>min</td>
</tr>
<tr>
<td>maxMeshTotalMemorySize</td>
<td>-</td>
<td>16384</td>
<td>min</td>
</tr>
<tr>
<td>maxMeshOutputVertices</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxMeshOutputPrimitives</td>
<td>-</td>
<td>256</td>
<td>min</td>
</tr>
<tr>
<td>maxMeshMultiviewViewCount</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>meshOutputPerVertexGranularity</td>
<td>-</td>
<td>-</td>
<td>implementatio dependent</td>
</tr>
<tr>
<td>meshOutputPerPrimitiveGranularity</td>
<td>-</td>
<td>-</td>
<td>implementatio dependent</td>
</tr>
<tr>
<td>maxTransformFeedbackStreams</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>maxTransformFeedbackBuffers</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------</td>
<td>----------------</td>
<td>------------</td>
</tr>
<tr>
<td>maxTransformFeedbackBufferSize</td>
<td>-</td>
<td>$2^{27}$</td>
<td>min</td>
</tr>
<tr>
<td>maxTransformFeedbackStreamDataSize</td>
<td>-</td>
<td>512</td>
<td>min</td>
</tr>
<tr>
<td>maxTransformFeedbackBufferDataSize</td>
<td>-</td>
<td>512</td>
<td>min</td>
</tr>
<tr>
<td>maxTransformFeedbackBufferDataStride</td>
<td>-</td>
<td>512</td>
<td>min</td>
</tr>
<tr>
<td>transformFeedbackQueries</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>transformFeedbackStreamsLinesTriangles</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>transformFeedbackRasterizationStreamSelect</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>transformFeedbackDraw</td>
<td>-</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>minFragmentDensityTexelSize</td>
<td>-</td>
<td>(1,1)</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentDensityTexelSize</td>
<td>-</td>
<td>(1,1)</td>
<td>min</td>
</tr>
<tr>
<td>fragmentDensityInvocations</td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>subsampledLoads</td>
<td>true</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>subsampledCoarseReconstructionEarlyAccess</td>
<td>false</td>
<td>false</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>maxSubsampledArrayLayers</td>
<td>2</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetSubsampledSamplers</td>
<td>1</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>VkPhysicalDeviceRayTracingPropertiesNV::shaderGroupHandleSize</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>VkPhysicalDeviceRayTracingPropertiesNV::maxRecursionDepth</td>
<td>-</td>
<td>31</td>
<td>min</td>
</tr>
<tr>
<td>VkPhysicalDeviceRayTracingPipelinePropertiesKHR::shaderGroupHandleSize</td>
<td>-</td>
<td>32</td>
<td>exact</td>
</tr>
<tr>
<td>VkPhysicalDeviceRayTracingPipelinePropertiesKHR::maxRayRecursionDepth</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>maxShaderGroupStride</td>
<td>-</td>
<td>4096</td>
<td>min</td>
</tr>
<tr>
<td>shaderGroupBaseAlignment</td>
<td>-</td>
<td>64</td>
<td>max</td>
</tr>
<tr>
<td>maxGeometryCount</td>
<td>-</td>
<td>$2^{24}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxInstanceCount</td>
<td>-</td>
<td>$2^{24}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxTriangleCount</td>
<td>-</td>
<td>$2^{29}$-1</td>
<td>min</td>
</tr>
<tr>
<td>maxPrimitiveCount</td>
<td>-</td>
<td>$2^{29}$-1</td>
<td>min</td>
</tr>
<tr>
<td>Limit</td>
<td>Unsupported Limit</td>
<td>Supported Limit</td>
<td>Limit Type¹</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>maxPerStageDescriptorAccelerationStructures</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxPerStageDescriptorUpdateAfterBindAccelerationStructures</td>
<td>-</td>
<td>500000³</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetAccelerationStructures</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxDescriptorSetUpdateAfterBindAccelerationStructures</td>
<td>-</td>
<td>500000³</td>
<td>min</td>
</tr>
<tr>
<td>minAccelerationStructureScratchOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>shaderGroupHandleCaptureReplaySize</td>
<td>-</td>
<td>64</td>
<td>max</td>
</tr>
<tr>
<td>maxRayDispatchInvocationCount</td>
<td>-</td>
<td>$2^{30}$</td>
<td>min</td>
</tr>
<tr>
<td>shaderGroupHandleAlignment</td>
<td>-</td>
<td>32</td>
<td>max</td>
</tr>
<tr>
<td>maxRayHitAttributeSize</td>
<td>-</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>maxTimelineSemaphoreValueDifference</td>
<td>-</td>
<td>$2^{31}$-1</td>
<td>min</td>
</tr>
<tr>
<td>lineSubPixelPrecisionBits</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxGraphicsShaderGroupCount</td>
<td>-</td>
<td>$2^{12}$</td>
<td>min</td>
</tr>
<tr>
<td>maxIndirectSequenceCount</td>
<td>-</td>
<td>$2^{20}$</td>
<td>min</td>
</tr>
<tr>
<td>maxIndirectCommandsTokenCount</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxIndirectCommandsStreamCount</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxIndirectCommandsTokenOffset</td>
<td>-</td>
<td>2047</td>
<td>min</td>
</tr>
<tr>
<td>maxIndirectCommandsStreamStride</td>
<td>-</td>
<td>2048</td>
<td>min</td>
</tr>
<tr>
<td>minSequencesCountBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minSequencesIndexBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minIndirectCommandsBufferOffsetAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>maxCustomBorderColorSamplers</td>
<td>-</td>
<td>32</td>
<td>min</td>
</tr>
<tr>
<td>robustStorageBufferAccessSizeAlignment</td>
<td>-</td>
<td>4</td>
<td>max</td>
</tr>
<tr>
<td>robustUniformBufferAccessSizeAlignment</td>
<td>-</td>
<td>256</td>
<td>max</td>
</tr>
<tr>
<td>minFragmentShadingRateAttachmentTexelSize</td>
<td>(0,0)</td>
<td>(32,32)</td>
<td>max</td>
</tr>
<tr>
<td>maxFragmentShadingRateAttachmentTexelSize</td>
<td>(0,0)</td>
<td>(8,8)</td>
<td>min</td>
</tr>
<tr>
<td>maxFragmentShadingRateAttachmentTexelSizeAspectRatio</td>
<td>0</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>primitiveFragmentShadingRateWithMultipleViewports</td>
<td>false</td>
<td>false</td>
<td>implementatio n-dependent</td>
</tr>
<tr>
<td>layeredShadingRateAttachments</td>
<td>false</td>
<td>false</td>
<td>implementatio n-dependent</td>
</tr>
<tr>
<td>fragmentShadingRateNonTrivialCombinerOps</td>
<td>-</td>
<td>false</td>
<td>implementatio n-dependent</td>
</tr>
</tbody>
</table>
| Limit                                              | Unsupported Limit | Supported Limit | Limit Type
|---------------------------------------------------|-------------------|-----------------|------------
| minFragmentSize                                   | -                 | (2,2)           | max        |
| maxFragmentSize                                   | -                 | (4,4)           | min        |
| maxFragmentSizeAspectRatio                       | -                 | 2               | min        |
| maxFragmentShadingRateCoverageSamples            | -                 | 16              | min        |
| maxFragmentShadingRateRasterizationSamples       | -                 | VK_SAMPLE_COUNT_4_BIT | min        |
| fragmentShadingRateWithShaderDepthStencil Writes | -                 | false           | implementation-dependent |
| fragmentShadingRateWithSampleMask                | -                 | false           | implementation-dependent |
| fragmentShadingRateWithShaderSampleMask          | -                 | false           | implementation-dependent |
| fragmentShadingRateWithConservativeRasterization | -                 | false           | implementation-dependent |
| fragmentShadingRateWithFragmentShaderInterlock   | -                 | false           | implementation-dependent |
| fragmentShadingRateWithCustomSampleLocations     | -                 | false           | implementation-dependent |
| fragmentShadingRateStrictMultiplyCombiner         | -                 | false           | implementation-dependent |
| maxFragmentShadingRateInvocationCount             | -                 | VK_SAMPLE_COUNT_4_BIT | min        |

1

The **Limit Type** column specifies the limit is either the minimum limit all implementations **must** support, the maximum limit all implementations **must** support, or the exact value all implementations **must** support. For bitmasks a minimum limit is the least bits all implementations **must** set, but they **may** have additional bits set beyond this minimum.

2

The **maxPerStageResources** **must** be at least the smallest of the following:

- the sum of the **maxPerStageDescriptorUniformBuffers**, **maxPerStageDescriptorStorageBuffers**, **maxPerStageDescriptorSampledImages**, **maxPerStageDescriptorStorageImages**, **maxPerStageDescriptorInputAttachments**, **maxColorAttachments** limits, or
- 128.

It **may** not be possible to reach this limit in every stage.

3

See **maxViewportDimensions** for the **required** relationship to other limits.
See `viewportBoundsRange` for the **required** relationship to other limits.

The values `minInterpolationOffset` and `maxInterpolationOffset` describe the closed interval of supported interpolation offsets: \([\text{minInterpolationOffset}, \text{maxInterpolationOffset}]\). The ULP is determined by `subPixelInterpolationOffsetBits`. If `subPixelInterpolationOffsetBits` is 4, this provides increments of \((1/2^4) = 0.0625\), and thus the range of supported interpolation offsets would be \([-0.5, 0.4375]\).

The point size ULP is determined by `pointSizeGranularity`. If the `pointSizeGranularity` is 0.125, the range of supported point sizes **must** be at least \([1.0, 63.875]\).

The line width ULP is determined by `lineWidthGranularity`. If the `lineWidthGranularity` is 0.0625, the range of supported line widths **must** be at least \([1.0, 7.9375]\).

The minimum `maxDescriptorSet` limit is \(n\) times the corresponding specification minimum `maxPerStageDescriptor` limit, where \(n\) is the number of shader stages supported by the `VkPhysicalDevice`. If all shader stages are supported, \(n = 6\) (vertex, tessellation control, tessellation evaluation, geometry, fragment, compute).

The `UpdateAfterBind` descriptor limits **must** each be greater than or equal to the corresponding non-`UpdateAfterBind` limit.

If the `VK_KHR_portability_subset` extension is enabled, the required minimum value of `maxVertexInputBindings` is 8.

### 42.2. Additional Multisampling Capabilities

To query additional multisampling capabilities which **may** be supported for a specific sample count, beyond the minimum capabilities described for `Limits` above, call:

```c
// Provided by VK_EXT_sample_locations
void vkGetPhysicalDeviceMultisamplePropertiesEXT(
    VkPhysicalDevice physicalDevice,
    VkSampleCountFlagBits samples,
    VkMultisamplePropertiesEXT* pMultisampleProperties);
```

- `physicalDevice` is the physical device from which to query the additional multisampling capabilities.
- `samples` is the sample count to query the capabilities for.
• `pMultisampleProperties` is a pointer to a `VkMultisamplePropertiesEXT` structure in which information about the additional multisampling capabilities specific to the sample count is returned.

### Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceMultisamplePropertiesEXT-physicalDevice-parameter
  physicalDevice must be a valid `VkPhysicalDevice` handle`
- `VUID-vkGetPhysicalDeviceMultisamplePropertiesEXT-samples-parameter
  samples must be a valid `VkSampleCountFlagBits` value`
- `VUID-vkGetPhysicalDeviceMultisamplePropertiesEXT-pMultisampleProperties-parameter
  pMultisampleProperties must be a valid pointer to a `VkMultisamplePropertiesEXT` structure`

The `VkMultisamplePropertiesEXT` structure is defined as

```c
// Provided by VK_EXT_sample_locations
typedef struct VkMultisamplePropertiesEXT {
  VkStructureType    sType;
  void*              pNext;
  VkExtent2D         maxSampleLocationGridSize;
} VkMultisamplePropertiesEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `maxSampleLocationGridSize` is the maximum size of the pixel grid in which sample locations can vary.

### Valid Usage (Implicit)

- `VUID-VkMultisamplePropertiesEXT-sType-sType
  sType must be `VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT`
- `VUID-VkMultisamplePropertiesEXT-pNext-pNext
  pNext must be `NULL`

If the sample count for which additional multisampling capabilities are requested using `vkGetPhysicalDeviceMultisamplePropertiesEXT` is set in `VkPhysicalDeviceSampleLocationsPropertiesEXT::sampleLocationSampleCounts` the width and height members of `VkMultisamplePropertiesEXT::maxSampleLocationGridSize` must be greater than or equal to the corresponding members of `VkPhysicalDeviceSampleLocationsPropertiesEXT::maxSampleLocationGridSize`, respectively, otherwise both members must be 0.
Chapter 43. Formats

Supported buffer and image formats may vary across implementations. A minimum set of format features are guaranteed, but others must be explicitly queried before use to ensure they are supported by the implementation.

The features for the set of formats (VkFormat) supported by the implementation are queried individually using the vkGetPhysicalDeviceFormatProperties command.

43.1. Format Definition

The following image formats can be passed to, and may be returned from Vulkan commands. The memory required to store each format is discussed with that format, and also summarized in the Representation and Texel Block Size section and the Compatible formats table.

```c
// Provided by VK_VERSION_1_0
typedef enum VkFormat {
    VK_FORMAT_UNDEFINED = 0,
    VK_FORMAT_R4G4_UNORM_PACK8 = 1,
    VK_FORMAT_R4G4B4A4_UNORM_PACK16 = 2,
    VK_FORMAT_B4G4R4A4_UNORM_PACK16 = 3,
    VK_FORMAT_R5G6B5_UNORM_PACK16 = 4,
    VK_FORMAT_B5G6R5_UNORM_PACK16 = 5,
    VK_FORMAT_R5G5B5A1_UNORM_PACK16 = 6,
    VK_FORMAT_B5G5R5A1_UNORM_PACK16 = 7,
    VK_FORMAT_A1R5G5B5_UNORM_PACK16 = 8,
    VK_FORMAT_R8_UNORM = 9,
    VK_FORMAT_R8_SNORM = 10,
    VK_FORMAT_R8_USCALED = 11,
    VK_FORMAT_R8_SSCALED = 12,
    VK_FORMAT_R8_UINT = 13,
    VK_FORMAT_R8_SINT = 14,
    VK_FORMAT_R8_SRGB = 15,
    VK_FORMAT_R8G8_UNORM = 16,
    VK_FORMAT_R8G8_SNORM = 17,
    VK_FORMAT_R8G8_USCALED = 18,
    VK_FORMAT_R8G8_SSCALED = 19,
    VK_FORMAT_R8G8_UINT = 20,
    VK_FORMAT_R8G8_SINT = 21,
    VK_FORMAT_R8G8_SRGB = 22,
    VK_FORMAT_R8G8B8_UNORM = 23,
    VK_FORMAT_R8G8B8_SNORM = 24,
    VK_FORMAT_R8G8B8_USCALED = 25,
    VK_FORMAT_R8G8B8_SSCALED = 26,
    VK_FORMAT_R8G8B8_UINT = 27,
    VK_FORMAT_R8G8B8_SINT = 28,
    VK_FORMAT_R8G8B8_SRGB = 29,
    VK_FORMAT_B8G8R8_UNORM = 30,
    VK_FORMAT_B8G8R8_SNORM = 31,

```

2260 | Chapter 43. Formats
| VK_FORMAT_B8G8R8_USCALED = 32, |
| VK_FORMAT_B8G8R8_SSCALED = 33, |
| VK_FORMAT_B8G8R8_UINT = 34, |
| VK_FORMAT_B8G8R8_SINT = 35, |
| VK_FORMAT_B8G8R8_SRGB = 36, |
| VK_FORMAT_R8G8B8A8_UNORM = 37, |
| VK_FORMAT_R8G8B8A8_SSCALED = 39, |
| VK_FORMAT_R8G8B8A8_UINT = 41, |
| VK_FORMAT_R8G8B8A8_SINT = 42, |
| VK_FORMAT_R8G8B8A8_SRGB = 43, |
| VK_FORMAT_B8G8R8A8_UNORM = 44, |
| VK_FORMAT_B8G8R8A8_SSCALED = 47, |
| VK_FORMAT_B8G8R8A8_UINT = 48, |
| VK_FORMAT_B8G8R8A8_SINT = 49, |
| VK_FORMAT_B8G8R8A8_SRGB = 50, |
| VK_FORMAT_A8B8G8R8_UNORM_PACK32 = 51, |
| VK_FORMAT_A8B8G8R8_SNORM_PACK32 = 53, |
| VK_FORMAT_A8B8G8R8_USCALED_PACK32 = 55, |
| VK_FORMAT_A8B8G8R8_SSCALED_PACK32 = 57, |
| VK_FORMAT_A8B8G8R8_UINT_PACK32 = 59, |
| VK_FORMAT_A8B8G8R8_SINT_PACK32 = 61, |
| VK_FORMAT_R16_SFLOAT = 76, |
| VK_FORMAT_R16_SINT = 75, |
| VK_FORMAT_R16_SINT = 82, |
VK_FORMAT_R16G16_SFLOAT = 83,
VK_FORMAT_R16G16B16_UNORM = 84,
VK_FORMAT_R16G16B16_SNORM = 85,
VK_FORMAT_R16G16B16_USCALED = 86,
VK_FORMAT_R16G16B16_SSCALED = 87,
VK_FORMAT_R16G16B16_UINT = 88,
VK_FORMAT_R16G16B16_SINT = 89,
VK_FORMAT_R16G16B16_SFLOAT = 90,
VK_FORMAT_R16G16B16A16_UNORM = 91,
VK_FORMAT_R16G16B16A16_SNORM = 92,
VK_FORMAT_R16G16B16A16_USCALED = 93,
VK_FORMAT_R16G16B16A16_SSCALED = 94,
VK_FORMAT_R16G16B16A16_UINT = 95,
VK_FORMAT_R16G16B16A16_SINT = 96,
VK_FORMAT_R16G16B16A16_SFLOAT = 97,
VK_FORMAT_R32_UINT = 98,
VK_FORMAT_R32_SINT = 99,
VK_FORMAT_R32_SFLOAT = 100,
VK_FORMAT_R32G32_UINT = 101,
VK_FORMAT_R32G32_SINT = 102,
VK_FORMAT_R32G32_SFLOAT = 103,
VK_FORMAT_R32G32B32_UINT = 104,
VK_FORMAT_R32G32B32_SINT = 105,
VK_FORMAT_R32G32B32_SFLOAT = 106,
VK_FORMAT_R32G32B32A32_UINT = 107,
VK_FORMAT_R32G32B32A32_SINT = 108,
VK_FORMAT_R32G32B32A32_SFLOAT = 109,
VK_FORMAT_R64_UINT = 110,
VK_FORMAT_R64_SINT = 111,
VK_FORMAT_R64_SFLOAT = 112,
VK_FORMAT_R64G64_UINT = 113,
VK_FORMAT_R64G64_SINT = 114,
VK_FORMAT_R64G64_SFLOAT = 115,
VK_FORMAT_R64G64B64_UINT = 116,
VK_FORMAT_R64G64B64_SINT = 117,
VK_FORMAT_R64G64B64_SFLOAT = 118,
VK_FORMAT_R64G64B64A64_UINT = 119,
VK_FORMAT_R64G64B64A64_SINT = 120,
VK_FORMAT_R64G64B64A64_SFLOAT = 121,
VK_FORMAT_B10G11R11_UFLOAT_PACK32 = 122,
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 = 123,
VK_FORMAT_D16_UNORM = 124,
VK_FORMAT_X8_D24_UNORM_PACK32 = 125,
VK_FORMAT_D32_SFLOAT = 126,
VK_FORMAT_S8_UINT = 127,
VK_FORMAT_D16_UNORM_S8_UINT = 128,
VK_FORMAT_D24_UNORM_S8_UINT = 129,
VK_FORMAT_D32_SFLOAT_S8_UINT = 130,
VK_FORMAT_BC1_RGB_UNORM_BLOCK = 131,
VK_FORMAT_BC1_RGB_SRGB_BLOCK = 132,
VK_FORMAT_BC1_RGBA_UNORM_BLOCK = 133,
VK_FORMAT_BC1_RGBA_SRGB_BLOCK = 134,
VK_FORMAT_BC2_UNORM_BLOCK = 135,
VK_FORMAT_BC2_SRGB_BLOCK = 136,
VK_FORMAT_BC3_UNORM_BLOCK = 137,
VK_FORMAT_BC3_SRGB_BLOCK = 138,
VK_FORMAT_BC4_UNORM_BLOCK = 139,
VK_FORMAT_BC4_SNORM_BLOCK = 140,
VK_FORMAT_BC5_UNORM_BLOCK = 141,
VK_FORMAT_BC5_SNORM_BLOCK = 142,
VK_FORMAT_BCGH_UNORM_BLOCK = 143,
VK_FORMAT_BCGH_SNORM_BLOCK = 144,
VK_FORMAT_BC7_UNORM_BLOCK = 145,
VK_FORMAT_BC7_SRGB_BLOCK = 146,
VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK = 147,
VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK = 148,
VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK = 149,
VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK = 150,
VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK = 151,
VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK = 152,
VK_FORMAT_EAC_R11_UNORM_BLOCK = 153,
VK_FORMAT_EAC_R11_SNORM_BLOCK = 154,
VK_FORMAT_EAC_R11G11_UNORM_BLOCK = 155,
VK_FORMAT_EAC_R11G11_SNORM_BLOCK = 156,
VK_FORMAT_ASTC_4x4_UNORM_BLOCK = 157,
VK_FORMAT_ASTC_4x4_SRGB_BLOCK = 158,
VK_FORMAT_ASTC_5x4_UNORM_BLOCK = 159,
VK_FORMAT_ASTC_5x4_SRGB_BLOCK = 160,
VK_FORMAT_ASTC_5x5_UNORM_BLOCK = 161,
VK_FORMAT_ASTC_5x5_SRGB_BLOCK = 162,
VK_FORMAT_ASTC_6x5_UNORM_BLOCK = 163,
VK_FORMAT_ASTC_6x5_SRGB_BLOCK = 164,
VK_FORMAT_ASTC_6x6_UNORM_BLOCK = 165,
VK_FORMAT_ASTC_6x6_SRGB_BLOCK = 166,
VK_FORMAT_ASTC_8x5_UNORM_BLOCK = 167,
VK_FORMAT_ASTC_8x5_SRGB_BLOCK = 168,
VK_FORMAT_ASTC_8x6_UNORM_BLOCK = 169,
VK_FORMAT_ASTC_8x6_SRGB_BLOCK = 170,
VK_FORMAT_ASTC_8x8_UNORM_BLOCK = 171,
VK_FORMAT_ASTC_8x8_SRGB_BLOCK = 172,
VK_FORMAT_ASTC_10x5_UNORM_BLOCK = 173,
VK_FORMAT_ASTC_10x5_SRGB_BLOCK = 174,
VK_FORMAT_ASTC_10x6_UNORM_BLOCK = 175,
VK_FORMAT_ASTC_10x6_SRGB_BLOCK = 176,
VK_FORMAT_ASTC_10x8_UNORM_BLOCK = 177,
VK_FORMAT_ASTC_10x8_SRGB_BLOCK = 178,
VK_FORMAT_ASTC_10x10_UNORM_BLOCK = 179,
VK_FORMAT_ASTC_10x10_SRGB_BLOCK = 180,
VK_FORMAT_ASTC_12x10_UNORM_BLOCK = 181,
VK_FORMAT_ASTC_12x10_SRGB_BLOCK = 182,
VK_FORMAT_ASTC_12x12_UNORM_BLOCK = 183,
VK_FORMAT_ASTC_12x12_SRGB_BLOCK = 184,
VK_FORMAT_B8G8R8_422_UNORM = 1000156001,

VK_FORMAT_B8G8R8_422_UNORM = 1000156002,

VK_FORMAT_B8G8R8_422_UNORM = 1000156003,

VK_FORMAT_B8G8R8_422_UNORM = 1000156004,

VK_FORMAT_B8G8R8_422_UNORM = 1000156005,

VK_FORMAT_B8G8R8_422_UNORM = 1000156006,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16 = 1000156025,
// Provided by VK_VERSION_1_1
VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16 = 1000156026,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16B16G16R16_422_UNORM = 1000156027,
// Provided by VK_VERSION_1_1
VK_FORMAT_B16G16R16G16_422_UNORM = 1000156028,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM = 1000156029,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16R16_2PLANE_420_UNORM = 1000156030,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM = 1000156031,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16R16_2PLANE_422_UNORM = 1000156032,
// Provided by VK_VERSION_1_1
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM = 1000156033,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG = 1000054000,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG = 1000054001,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG = 1000054002,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG = 1000054003,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG = 1000054004,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG = 1000054005,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG = 1000054006,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG = 1000054007,
// Provided by VK_IMG_format_pvrtc
VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT = 1000066000,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT = 1000066001,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT = 1000066002,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT = 1000066003,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT = 1000066004,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT = 1000066005,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT = 1000066006,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT = 1000066007,
// Provided by VK_EXT_texture_compression_astc_hdr
VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT = 1000066008,
VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT = 1000066009,
VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT = 1000066010,
VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT = 1000066011,
VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT = 1000066012,
VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT = 1000066013,
VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT = 1000330000,
VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT = 1000330001,
VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT = 1000330002,
VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT = 1000330003,
VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT = 1000340000,
VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT = 1000340001,
VK_FORMAT_G8B8G8R8_422_UNORM_KHR = VK_FORMAT_G8B8G8R8_422_UNORM,
VK_FORMAT_B8G8R8G8_422_UNORM_KHR = VK_FORMAT_B8G8R8G8_422_UNORM,
VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM_KHR = VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM,
VK_FORMAT_G8_B8R8_2PLANE_420_UNORM_KHR = VK_FORMAT_G8_B8R8_2PLANE_420_UNORM,
VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM_KHR = VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM,
VK_FORMAT_R10X6_UNORM_PACK16_KHR = VK_FORMAT_R10X6_UNORM_PACK16,
VK_FORMAT_R10X6G10X6_UNORM_2PACK16_KHR = VK_FORMAT_R10X6G10X6_UNORM_2PACK16,
VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16_KHR = VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16,
VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16_KHR =
  VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16_KHR =
  VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16_KHR =
  VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16_KHR =
  VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_R12X4_UNIFORM_2PACK16_KHR = VK_FORMAT_R12X4_UNIFORM_2PACK16,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_R12X4G12X4_UNIFORM_2PACK16_KHR = VK_FORMAT_R12X4G12X4_UNIFORM_2PACK16,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_R12X4G12X4B12X4A12X4_UNIFORM_4PACK16_KHR =
  VK_FORMAT_R12X4G12X4B12X4A12X4_UNIFORM_4PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_KHR,
  // Provided by VK_KHR_sampler_yccbcr_conversion
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_KHR =
  VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_KHR,
VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM_KHR = VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM,
// Provided by VK_KHR_sampler_ycbcr_conversion
VK_FORMAT_G16_B16R16_2PLANE_422_UNORM_KHR = VK_FORMAT_G16_B16R16_2PLANE_422_UNORM,
// Provided by VK_KHR_sampler_ycbcr_conversion
VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM_KHR = VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM,
} VkFormat;

- **VK_FORMAT_UNDEFINED** specifies that the format is not specified.
- **VK_FORMAT_R4G4_UNORM_PACK8** specifies a two-component, 8-bit packed unsigned normalized format that has a 4-bit R component in bits 4..7, and a 4-bit G component in bits 0..3.
- **VK_FORMAT_R4G4B4A4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit R component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit B component in bits 4..7, and a 4-bit A component in bits 0..3.
- **VK_FORMAT_B4G4R4A4_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit B component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit R component in bits 4..7, and a 4-bit A component in bits 0..3.
- **VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit A component in bits 12..15, a 4-bit R component in bits 8..11, a 4-bit G component in bits 4..7, and a 4-bit B component in bits 0..3.
- **VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT** specifies a four-component, 16-bit packed unsigned normalized format that has a 4-bit A component in bits 12..15, a 4-bit B component in bits 8..11, a 4-bit G component in bits 4..7, and a 4-bit R component in bits 0..3.
- **VK_FORMAT_R5G5B5A1_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit B component in bits 1..5, and a 1-bit A component in bit 0.
- **VK_FORMAT_B5G5R5A1_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit R component in bits 1..5, and a 1-bit A component in bit 0.
- **VK_FORMAT_A1R5G5B5_UNORM_PACK16** specifies a four-component, 16-bit packed unsigned normalized format that has a 1-bit A component in bit 15, a 5-bit R component in bits 10..14, a 5-bit G component in bits 5..9, and a 5-bit B component in bits 0..4.
- **VK_FORMAT_R8_UNORM** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component.
- **VK_FORMAT_R8_SNORM** specifies a one-component, 8-bit signed normalized format that has a single 8-bit R component.
• **VK_FORMAT_R8_USCALED** specifies a one-component, 8-bit unsigned scaled integer format that has a single 8-bit R component.

• **VK_FORMAT_R8_SSCALED** specifies a one-component, 8-bit signed scaled integer format that has a single 8-bit R component.

• **VK_FORMAT_R8_UINT** specifies a one-component, 8-bit unsigned integer format that has a single 8-bit R component.

• **VK_FORMAT_R8_SINT** specifies a one-component, 8-bit signed integer format that has a single 8-bit R component.

• **VK_FORMAT_R8_SRGB** specifies a one-component, 8-bit unsigned normalized format that has a single 8-bit R component stored with sRGB nonlinear encoding.

• **VK_FORMAT_R8G8_UNORM** specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

• **VK_FORMAT_R8G8_SNORM** specifies a two-component, 16-bit signed normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

• **VK_FORMAT_R8G8_USCALED** specifies a two-component, 16-bit unsigned scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

• **VK_FORMAT_R8G8_SSCALED** specifies a two-component, 16-bit signed scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

• **VK_FORMAT_R8G8_UINT** specifies a two-component, 16-bit unsigned integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

• **VK_FORMAT_R8G8_SINT** specifies a two-component, 16-bit signed integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

• **VK_FORMAT_R8G8_SRGB** specifies a two-component, 16-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, and an 8-bit G component stored with sRGB nonlinear encoding in byte 1.

• **VK_FORMAT_R8G8B8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_SSCALED** specifies a three-component, 24-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

• **VK_FORMAT_R8G8B8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 0, and an 8-bit B component in byte 2.
• **VK_FORMAT_B8G8R8_UNORM** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_SNORM** specifies a three-component, 24-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_USCALED** specifies a three-component, 24-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_SSCALED** specifies a three-component, 24-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_UINT** specifies a three-component, 24-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_SINT** specifies a three-component, 24-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

• **VK_FORMAT_B8G8R8_SRGB** specifies a three-component, 24-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit R component stored with sRGB nonlinear encoding in byte 2.

• **VK_FORMAT_R8G8B8A8_UNORM** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_R8G8B8A8_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_R8G8B8A8_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_R8G8B8A8_SSCALED** specifies a four-component, 32-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_R8G8B8A8_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_R8G8B8A8_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_R8G8B8A8_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit B component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.
• **VK_FORMAT_B8G8R8A8_UNORM** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_B8G8R8A8_SNORM** specifies a four-component, 32-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_B8G8R8A8_USCALED** specifies a four-component, 32-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_B8G8R8A8_SSCALED** specifies a four-component, 32-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_B8G8R8A8_UINT** specifies a four-component, 32-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_B8G8R8A8_SINT** specifies a four-component, 32-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_B8G8R8A8_SRGB** specifies a four-component, 32-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit R component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.

• **VK_FORMAT_A8B8G8R8_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

• **VK_FORMAT_A8B8G8R8_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

• **VK_FORMAT_A8B8G8R8_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

• **VK_FORMAT_A8B8G8R8_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

• **VK_FORMAT_A8B8G8R8_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

• **VK_FORMAT_A8B8G8R8_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

• **VK_FORMAT_A8B8G8R8_SRGB_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component stored with sRGB nonlinear encoding in bits 16..23, an 8-bit G component stored with sRGB nonlinear encoding in bits 8..15, and an 8-bit R component stored with sRGB nonlinear encoding in bits 0..7.
bits 8..15, and an 8-bit R component stored with sRGB nonlinear encoding in bits 0..7.

- **VK_FORMAT_A2R10G10B10_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2R10G10B10_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_UNORM_PACK32** specifies a four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_SNORM_PACK32** specifies a four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_USCALED_PACK32** specifies a four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_SSCALED_PACK32** specifies a four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_UINT_PACK32** specifies a four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_A2B10G10R10_SINT_PACK32** specifies a four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

- **VK_FORMAT_R16_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit R component.

- **VK_FORMAT_R16_SNORM** specifies a one-component, 16-bit signed normalized format that has a single 16-bit R component.

- **VK_FORMAT_R16_USCALED** specifies a one-component, 16-bit unsigned scaled integer format that has
a single 16-bit R component.

- **VK_FORMAT_R16_SSCALED** specifies a one-component, 16-bit signed scaled integer format that has a single 16-bit R component.
- **VK_FORMAT_R16_UINT** specifies a one-component, 16-bit unsigned integer format that has a single 16-bit R component.
- **VK_FORMAT_R16_SINT** specifies a one-component, 16-bit signed integer format that has a single 16-bit R component.
- **VK_FORMAT_R16_SFLOAT** specifies a one-component, 16-bit signed floating-point format that has a single 16-bit R component.

- **VK_FORMAT_R16G16_UNORM** specifies a two-component, 32-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK_FORMAT_R16G16_SNORM** specifies a two-component, 32-bit signed normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK_FORMAT_R16G16_USCALED** specifies a two-component, 32-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK_FORMAT_R16G16_SSCALED** specifies a two-component, 32-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK_FORMAT_R16G16_UINT** specifies a two-component, 32-bit unsigned integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK_FORMAT_R16G16_SINT** specifies a two-component, 32-bit signed integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.
- **VK_FORMAT_R16G16_SFLOAT** specifies a two-component, 32-bit signed floating-point format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

- **VK_FORMAT_R16G16B16_UNORM** specifies a three-component, 48-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK_FORMAT_R16G16B16_SNORM** specifies a three-component, 48-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK_FORMAT_R16G16B16_USCALED** specifies a three-component, 48-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK_FORMAT_R16G16B16_SSCALED** specifies a three-component, 48-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK_FORMAT_R16G16B16_UINT** specifies a three-component, 48-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
- **VK_FORMAT_R16G16B16_SINT** specifies a three-component, 48-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.
• **VK_FORMAT_R16G16B16_SFLOAT** specifies a three-component, 48-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

• **VK_FORMAT_R16G16B16A16_UNORM** specifies a four-component, 64-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R16G16B16A16_SNORM** specifies a four-component, 64-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R16G16B16A16_USCALED** specifies a four-component, 64-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R16G16B16A16_SSCALED** specifies a four-component, 64-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R16G16B16A16_UINT** specifies a four-component, 64-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R16G16B16A16_SINT** specifies a four-component, 64-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R16G16B16A16_SFLOAT** specifies a four-component, 64-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

• **VK_FORMAT_R32_UINT** specifies a one-component, 32-bit unsigned integer format that has a single 32-bit R component.

• **VK_FORMAT_R32_SINT** specifies a one-component, 32-bit signed integer format that has a single 32-bit R component.

• **VK_FORMAT_R32_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has a single 32-bit R component.

• **VK_FORMAT_R32G32_UINT** specifies a two-component, 64-bit unsigned integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

• **VK_FORMAT_R32G32_SINT** specifies a two-component, 64-bit signed integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

• **VK_FORMAT_R32G32_SFLOAT** specifies a two-component, 64-bit signed floating-point format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

• **VK_FORMAT_R32G32B32_UINT** specifies a three-component, 96-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

• **VK_FORMAT_R32G32B32_SINT** specifies a three-component, 96-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.
- **VK_FORMAT_R32G32B32_SFLOAT** specifies a three-component, 96-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

- **VK_FORMAT_R32G32B32A32_UINT** specifies a four-component, 128-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

- **VK_FORMAT_R32G32B32A32_SINT** specifies a four-component, 128-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

- **VK_FORMAT_R32G32B32A32_SFLOAT** specifies a four-component, 128-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

- **VK_FORMAT_R64_UINT** specifies a one-component, 64-bit unsigned integer format that has a single 64-bit R component.

- **VK_FORMAT_R64_SINT** specifies a one-component, 64-bit signed integer format that has a single 64-bit R component.

- **VK_FORMAT_R64_SFLOAT** specifies a one-component, 64-bit signed floating-point format that has a single 64-bit R component.

- **VK_FORMAT_R64G64_UINT** specifies a two-component, 128-bit unsigned integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

- **VK_FORMAT_R64G64_SINT** specifies a two-component, 128-bit signed integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

- **VK_FORMAT_R64G64_SFLOAT** specifies a two-component, 128-bit signed floating-point format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

- **VK_FORMAT_R64G64B64_UINT** specifies a three-component, 192-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

- **VK_FORMAT_R64G64B64_SINT** specifies a three-component, 192-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

- **VK_FORMAT_R64G64B64_SFLOAT** specifies a three-component, 192-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

- **VK_FORMAT_R64G64B64A64_UINT** specifies a four-component, 256-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

- **VK_FORMAT_R64G64B64A64_SINT** specifies a four-component, 256-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

- **VK_FORMAT_R64G64B64A64_SFLOAT** specifies a four-component, 256-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.
• **VK_FORMAT_B10G11R11_UFLOAT_PACK32** specifies a three-component, 32-bit packed unsigned floating-point format that has a 10-bit B component in bits 22..31, an 11-bit G component in bits 11..21, an 11-bit R component in bits 0..10. See [Unsigned 10-Bit Floating-Point Numbers](#) and [Unsigned 11-Bit Floating-Point Numbers](#).

• **VK_FORMAT_E5B9G9R9_UFLOAT_PACK32** specifies a three-component, 32-bit packed unsigned floating-point format that has a 5-bit shared exponent in bits 27..31, a 9-bit B component mantissa in bits 18..26, a 9-bit G component mantissa in bits 9..17, and a 9-bit R component mantissa in bits 0..8.

• **VK_FORMAT_D16_UNORM** specifies a one-component, 16-bit unsigned normalized format that has a single 16-bit depth component.

• **VK_FORMAT_X8_D24_UNORM_PACK32** specifies a two-component, 32-bit format that has 24 unsigned normalized bits in the depth component and, optionally, 8 bits that are unused.

• **VK_FORMAT_D32_SFLOAT** specifies a one-component, 32-bit signed floating-point format that has 32 bits in the depth component.

• **VK_FORMAT_S8_UINT** specifies a one-component, 8-bit unsigned integer format that has 8 bits in the stencil component.

• **VK_FORMAT_D16_UNORM_S8_UINT** specifies a two-component, 24-bit format that has 16 unsigned normalized bits in the depth component and 8 unsigned integer bits in the stencil component.

• **VK_FORMAT_D24_UNORM_S8_UINT** specifies a two-component, 32-bit packed format that has 8 unsigned integer bits in the stencil component, and 24 unsigned normalized bits in the depth component.

• **VK_FORMAT_D32_SFLOAT_S8_UINT** specifies a two-component format that has 32 signed float bits in the depth component and 8 unsigned integer bits in the stencil component. There are optionally: 24 bits that are unused.

• **VK_FORMAT_BC1_RGB_UNORM_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

• **VK_FORMAT_BC1_RGB_SRGB_BLOCK** specifies a three-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.

• **VK_FORMAT_BC1_RGBA_UNORM_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

• **VK_FORMAT_BC1_RGBA_SRGB_BLOCK** specifies a four-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

• **VK_FORMAT_BC2_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

• **VK_FORMAT_BC2_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding.
• **VK_FORMAT_BC3_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

• **VK_FORMAT_BC3_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding.

• **VK_FORMAT_BC4_UNORM_BLOCK** specifies a one-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized red texel data.

• **VK_FORMAT_BC4_SNORM_BLOCK** specifies a one-component, block-compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of signed normalized red texel data.

• **VK_FORMAT_BC5_UNORM_BLOCK** specifies a two-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

• **VK_FORMAT_BC5_SNORM_BLOCK** specifies a two-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

• **VK_FORMAT_BC6H_UFLOAT_BLOCK** specifies a three-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned floating-point RGB texel data.

• **VK_FORMAT_BC6H_SFLOAT_BLOCK** specifies a three-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed floating-point RGB texel data.

• **VK_FORMAT_BC7_UNORM_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.

• **VK_FORMAT_BC7_SRGB_BLOCK** specifies a four-component, block-compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

• **VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK** specifies a three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data. This format has no alpha and is considered opaque.

• **VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK** specifies a three-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.

• **VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data, and provides 1 bit of alpha.

• **VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGB texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

• **VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.
RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

- **VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK** specifies a four-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values with sRGB nonlinear encoding applied.

- **VK_FORMAT_EAC_R11_UNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized red texel data.

- **VK_FORMAT_EAC_R11_SNORM_BLOCK** specifies a one-component, ETC2 compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of signed normalized red texel data.

- **VK_FORMAT_EAC_R11G11_UNORM_BLOCK** specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

- **VK_FORMAT_EAC_R11G11_SNORM_BLOCK** specifies a two-component, ETC2 compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed normalized RG texel data with the first 64 bits encoding red values followed by 64 bits encoding green values.

- **VK_FORMAT_ASTC_4x4_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_4x4_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 4×4 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_5x4_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_5x4_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×4 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_5x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_5x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 5×5 rectangle of signed floating-point RGBA texel data.
RGBA texel data.

- **VK_FORMAT_ASTC_6x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_6x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×5 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_6x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_6x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 6×6 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_8x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_8x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 8×5 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_8x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_8x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 8×6 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_8x8_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×8 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_8x8_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes an 8×8 rectangle of unsigned normalized RGBA texel data.
texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 8×8 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x5_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×5 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x5_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×5 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×5 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x6_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×6 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x6_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×6 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×6 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x8_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×8 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x8_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×8 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×8 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_10x10_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×10 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_10x10_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 10×10 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_12x10_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of unsigned normalized RGBA texel data.
- **VK_FORMAT_ASTC_12x10_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×10 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_ASTC_12x12_UNORM_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_ASTC_12x12_SRGB_BLOCK** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT** specifies a four-component, ASTC compressed format where each 128-bit compressed texel block encodes a 12×12 rectangle of signed floating-point RGBA texel data.

- **VK_FORMAT_G8B8G8R8_422_UNORM** specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit G component for the even i coordinate in byte 0, an 8-bit B component in byte 1, an 8-bit G component for the odd i coordinate in byte 2, and an 8-bit R component in byte 3. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_B8G8R8G8_422_UNORM** specifies a four-component, 32-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has an 8-bit B component in byte 0, an 8-bit G component for the even i coordinate in byte 1, an 8-bit R component in byte 2, and an 8-bit G component for the odd i coordinate in byte 3. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( \lfloor i_G \times 0.5 \rfloor = i_R \) and \( \lfloor i_G \times 0.5 \rfloor = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using **VK_IMAGE_ASPECT_PLANE_0_BIT** for the G plane, **VK_IMAGE_ASPECT_PLANE_1_BIT** for the B plane, and **VK_IMAGE_ASPECT_PLANE_2_BIT** for the R plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G8_B8R8_2PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that
has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for which \( l_G \times 0.5 = i_B = i_R \) and \( j_G \times 0.5 = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width and height that is a multiple of two.

- `VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM` specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( l_G \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width that is a multiple of two.

- `VK_FORMAT_G8_B8R8_2PLANE_422_UNORM` specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( l_G \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width that is a multiple of two.

- `VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM` specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, an 8-bit B component in plane 1, and an 8-bit R component in plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- `VK_FORMAT_R10X6_UNORM_PACK16` specifies a one-component, 16-bit unsigned normalized format that has a single 10-bit R component in the top 10 bits of a 16-bit word, with the bottom 6 bits unused.

- `VK_FORMAT_R10X6G10X6_UNORM_2PACK16` specifies a two-component, 32-bit unsigned normalized format that has a 10-bit R component in the top 10 bits of the word in bytes 0..1, and a 10-bit G component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word unused.

- `VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16` specifies a four-component, 64-bit unsigned normalized format that has a 10-bit R component in the top 10 bits of the word in bytes 0..1, a 10-bit G component in the top 10 bits of the word in bytes 2..3, a 10-bit B component in the top 10 bits of the word in bytes 4..5, and a 10-bit A component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused.

- `VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16` specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i
coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit G component for the even $i$ coordinate in the top 10 bits of the word in bytes 0..1, a 10-bit B component in the top 10 bits of the word in bytes 2..3, a 10-bit G component for the odd $i$ coordinate in the top 10 bits of the word in bytes 4..5, and a 10-bit R component in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a $2 \times 1$ compressed texel block.

- **VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a $2 \times 1$ rectangle of unsigned normalized RGB texel data. One G value is present at each $i$ coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 10-bit B component in the top 10 bits of the word in bytes 0..1, a 10-bit G component for the even $i$ coordinate in the top 10 bits of the word in bytes 2..3, a 10-bit R component in the top 10 bits of the word in bytes 4..5, and a 10-bit G component for the odd $i$ coordinate in the top 10 bits of the word in bytes 6..7, with the bottom 6 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a $2 \times 1$ compressed texel block.

- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multiplanar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which $\lfloor G \times 0.5 \rfloor = i_B = i_R$ and $\lfloor G \times 0.5 \rfloor = j_B = j_R$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G10X6_B10X6_R10X6_2PLANE_420_UNORM_3PACK16** specifies an unsigned normalized multiplanar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word unused. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for which $\lfloor G \times 0.5 \rfloor = i_B = i_R$ and $\lfloor G \times 0.5 \rfloor = j_B = j_R$. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16** specifies an unsigned normalized multiplanar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which $\lfloor G \times 0.5 \rfloor = i_B = i_R$. The location of each
plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width that is a multiple of two.

- **`VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16`** specifies an unsigned normalized multiplanar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 10-bit B component in the top 10 bits of the word in bytes 0..1, and a 10-bit R component in the top 10 bits of the word in bytes 2..3, with the bottom 6 bits of each word unused. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( bG \times 0.5J = bR = iR \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width that is a multiple of two.

- **`VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16`** specifies an unsigned normalized multiplanar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, a 10-bit B component in the top 10 bits of each 16-bit word of plane 1, and a 10-bit R component in the top 10 bits of each 16-bit word of plane 2, with the bottom 6 bits of each word unused. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- **`VK_FORMAT_R12X4_UNORM_PACK16`** specifies a one-component, 16-bit unsigned normalized format that has a single 12-bit R component in the top 12 bits of a 16-bit word, with the bottom 4 bits unused.

- **`VK_FORMAT_R12X4G12X4_UNORM_2PACK16`** specifies a two-component, 32-bit unsigned normalized format that has a 12-bit R component in the top 12 bits of the word in bytes 0..1, and a 12-bit G component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word unused.

- **`VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16`** specifies a four-component, 64-bit unsigned normalized format that has a 12-bit R component in the top 12 bits of the word in bytes 0..1, a 12-bit G component in the top 12 bits of the word in bytes 2..3, a 12-bit B component in the top 12 bits of the word in bytes 4..5, and a 12-bit A component in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused.

- **`VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16`** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit G component for the even \( i \) coordinate in the top 12 bits of the word in bytes 0..1, a 12-bit B component in the top 12 bits of the word in bytes 2..3, a 12-bit G component for the odd \( i \) coordinate in the top 12 bits of the word in bytes 4..5, and a 12-bit R component in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.
• VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16 specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each i coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 12-bit B component in the top 12 bits of the word in bytes 0..1, a 12-bit G component for the even i coordinate in the top 12 bits of the word in bytes 2..3, a 12-bit R component in the top 12 bits of the word in bytes 4..5, and a 12-bit G component for the odd i coordinate in the top 12 bits of the word in bytes 6..7, with the bottom 4 bits of each word unused. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

• VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16 specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word unused. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B component is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \) and \( j_G \times 0.5 = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, VK_IMAGE_ASPECT_PLANE_1_BIT for the B plane, and VK_IMAGE_ASPECT_PLANE_2_BIT for the R plane. This format only supports images with a width that is a multiple of two.

• VK_FORMAT_G12X4_B12X4_R12X4_2PLANE_420_UNORM_3PACK16 specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word unused. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \) and \( j_G \times 0.5 = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, and VK_IMAGE_ASPECT_PLANE_1_BIT for the BR plane. This format only supports images with a width and height that is a multiple of two.

• VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16 specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word unused. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, VK_IMAGE_ASPECT_PLANE_1_BIT for the B plane, and VK_IMAGE_ASPECT_PLANE_2_BIT for the R plane. This format only supports images with a width that is a multiple of two.

• VK_FORMAT_G12X4_B12X4_R12X4_2PLANE_422_UNORM_3PACK16 specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, with the bottom 4 bits of each word unused. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( i_G \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via vkGetImageSubresourceLayout, using VK_IMAGE_ASPECT_PLANE_0_BIT for the G plane, VK_IMAGE_ASPECT_PLANE_1_BIT for the B plane, and VK_IMAGE_ASPECT_PLANE_2_BIT for the R plane. This format only supports images with a width that is a multiple of two.
the bottom 4 bits of each word unused. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( i_C \times 0.5 = i_B = i_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16** specifies an unsigned normalized multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, a 12-bit B component in the top 12 bits of each 16-bit word of plane 1, and a 12-bit R component in the top 12 bits of each 16-bit word of plane 2, with the bottom 4 bits of each word unused. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane.

- **VK_FORMAT_G16B16G16R16_422_UNORM** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 16-bit G component for the even \( i \) coordinate in the word in bytes 0..1, a 16-bit B component in the word in bytes 2..3, a 16-bit G component for the odd \( i \) coordinate in the word in bytes 4..5, and a 16-bit R component in the word in bytes 6..7. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_B16G16R16G16_422_UNORM** specifies a four-component, 64-bit format containing a pair of G components, an R component, and a B component, collectively encoding a 2×1 rectangle of unsigned normalized RGB texel data. One G value is present at each \( i \) coordinate, with the B and R values shared across both G values and thus recorded at half the horizontal resolution of the image. This format has a 16-bit B component in the word in bytes 0..1, a 16-bit G component for the even \( i \) coordinate in the word in bytes 2..3, a 16-bit R component in the word in bytes 4..5, and a 16-bit G component for the odd \( i \) coordinate in the word in bytes 6..7. This format only supports images with a width that is a multiple of two. For the purposes of the constraints on copy extents, this format is treated as a compressed format with a 2×1 compressed texel block.

- **VK_FORMAT_G16_B16R16_2PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal and vertical dimensions of the R and B planes are halved relative to the image dimensions, and each R and B value is shared with the G components for which \( j_C \times 0.5 = j_B = j_R \). The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, `VK_IMAGE_ASPECT_PLANE_1_BIT` for the B plane, and `VK_IMAGE_ASPECT_PLANE_2_BIT` for the R plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G16_B16R16_2PLANE_420_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. The horizontal and vertical dimensions of the BR plane are halved relative to the image dimensions, and each R and B value is shared with the G components for
which \( l_G \times 0.5 = I_B = I_R \) and \( j_G \times 0.5 = J_B = J_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane. This format only supports images with a width and height that is a multiple of two.

- **VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. The horizontal dimension of the R and B plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( l_G \times 0.5 = I_B = I_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G16_B16R16_2PLANE_422_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. The horizontal dimension of the BR plane is halved relative to the image dimensions, and each R and B value is shared with the G components for which \( l_G \times 0.5 = I_B = I_R \). The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane. This format only supports images with a width that is a multiple of two.

- **VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, a 16-bit B component in each 16-bit word of plane 1, and a 16-bit R component in each 16-bit word of plane 2. Each plane has the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the B plane, and \( \text{VK_IMAGE_ASPECT_PLANE_2_BIT} \) for the R plane.

- **VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT** specifies an unsigned normalized multi-planar format that has an 8-bit G component in plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane.

- **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT** specifies an unsigned normalized multi-planar format that has a 10-bit G component in the top 10 bits of each 16-bit word of plane 0, and a two-component, 16-bit BR plane 1 consisting of an 8-bit B component in byte 0 and an 8-bit R component in byte 1. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via \( \text{vkGetImageSubresourceLayout} \), using \( \text{VK_IMAGE_ASPECT_PLANE_0_BIT} \) for the G plane, and \( \text{VK_IMAGE_ASPECT_PLANE_1_BIT} \) for the BR plane.

- **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT** specifies an unsigned normalized
multi-planar format that has a 12-bit G component in the top 12 bits of each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 12-bit B component in the top 12 bits of the word in bytes 0..1, and a 12-bit R component in the top 12 bits of the word in bytes 2..3, the bottom 4 bits of each word unused. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

- **VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT** specifies an unsigned normalized multi-planar format that has a 16-bit G component in each 16-bit word of plane 0, and a two-component, 32-bit BR plane 1 consisting of a 16-bit B component in the word in bytes 0..1, and a 16-bit R component in the word in bytes 2..3. Both planes have the same dimensions and each R, G and B component contributes to a single texel. The location of each plane when this image is in linear layout can be determined via `vkGetImageSubresourceLayout`, using `VK_IMAGE_ASPECT_PLANE_0_BIT` for the G plane, and `VK_IMAGE_ASPECT_PLANE_1_BIT` for the BR plane.

- **VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an 8×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an 8×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an 8×4 rectangle of unsigned normalized RGBA texel data.

- **VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a 8×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes an 8×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

- **VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG** specifies a four-component, PVRTC compressed format where each 64-bit compressed texel block encodes a 4×4 rectangle of unsigned normalized RGBA texel data with sRGB nonlinear encoding applied to the RGB components.

### 43.1.1. Compatible formats of planes of multi-planar formats

Individual planes of multi-planar formats are compatible with single-plane formats if they occupy the same number of bits per texel block. In the following table, individual planes of a multi-planar format are compatible with the format listed against the relevant plane index for that multi-planar...
format, and any format compatible with the listed single-plane format according to Format Compatibility Classes.

### Table 55. Plane Format Compatibility Table

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width $w$ of the plane with the largest dimensions</th>
<th>Height relative to the height $h$ of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM$</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>0</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>1</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>$VK_FORMAT_G8_B8_R8_2PLANE_420_UNORM$</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>0</td>
<td>$VK_FORMAT_R8G8_UNORM$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>$VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM$</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>0</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>$VK_FORMAT_R8_UNORM$</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>$VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16$</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>0</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>1</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>$VK_FORMAT_G10X6_B10X6_R10X6_2PLANE_420_UNORM_3PACK16$</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>0</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>1</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h/2$</td>
</tr>
<tr>
<td>$VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16$</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w$</td>
<td>$h$</td>
</tr>
<tr>
<td>0</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>1</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>2</td>
<td>$VK_FORMAT_R10X6_UNORM_PACK16$</td>
<td>$w/2$</td>
<td>$h$</td>
</tr>
<tr>
<td>Plane</td>
<td>Compatible format for plane</td>
<td>Width relative to the width ( w ) of the plane with the largest dimensions</td>
<td>Height relative to the height ( h ) of the plane with the largest dimensions</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6G10X6_UNORM_2PACK16</td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R10X6G10X6_UNORM_2PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R10X6_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R12X4_UNORM_PACK16</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w/2 )</td>
<td>( h/2 )</td>
</tr>
<tr>
<td>Plane</td>
<td>Compatible format for plane</td>
<td>Width relative to the width ( w ) of the plane with the largest dimensions</td>
<td>Height relative to the height ( h ) of the plane with the largest dimensions</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16_UNORM</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
<tr>
<td>2</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w/2 )</td>
<td>( h )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plane</th>
<th>Compatible format for plane</th>
<th>Width relative to the width ( w ) of the plane with the largest dimensions</th>
<th>Height relative to the height ( h ) of the plane with the largest dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>0</td>
<td>VK_FORMAT_R16G16_UNORM</td>
<td>( w )</td>
<td>( h )</td>
</tr>
<tr>
<td>1</td>
<td>VK_FORMAT_R16G16UNORM</td>
<td>( w )</td>
<td>( h )</td>
</tr>
</tbody>
</table>

43.1.2. Packed Formats

For the purposes of address alignment when accessing buffer memory containing vertex attribute or texel data, the following formats are considered packed - whole texels or attributes are stored in bitfields of a single 8-, 16-, or 32-bit fundamental data type.

- **Packed into 8-bit data types:**
  - VK_FORMAT_R4G4_UNORM_PACK8
- **Packed into 16-bit data types:**
  - VK_FORMAT_R4G4B4A4_UNORM_PACK16
  - VK_FORMAT_B4G4R4A4_UNORM_PACK16
43.1.3. Identification of Formats

A “format” is represented by a single enum value. The name of a format is usually built up by using the following pattern:
The component-format indicates either the size of the R, G, B, and A components (if they are present) in the case of a color format, or the size of the depth (D) and stencil (S) components (if they are present) in the case of a depth/stencil format (see below). An X indicates a component that is unused, but may be present for padding.
Table 56. Interpretation of Numeric Format

<table>
<thead>
<tr>
<th>Numeric format</th>
<th>SPIR-V Sampled Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNORM</td>
<td>OpTypeFloat</td>
<td>The components are unsigned normalized values in the range [0,1]</td>
</tr>
<tr>
<td>SNORM</td>
<td>OpTypeFloat</td>
<td>The components are signed normalized values in the range [-1,1]</td>
</tr>
<tr>
<td>USCALED</td>
<td>OpTypeFloat</td>
<td>The components are unsigned integer values that get converted to floating-point in the range [0,2^n-1]</td>
</tr>
<tr>
<td>SScaled</td>
<td>OpTypeFloat</td>
<td>The components are signed integer values that get converted to floating-point in the range [-2^{n-1},2^{n-1}-1]</td>
</tr>
<tr>
<td>UINT</td>
<td>OpTypeInt</td>
<td>The components are unsigned integer values in the range [0,2^n-1]</td>
</tr>
<tr>
<td>SINT</td>
<td>OpTypeInt</td>
<td>The components are signed integer values in the range [-2^{n-1},2^{n-1}-1]</td>
</tr>
<tr>
<td>UFLOAT</td>
<td>OpTypeFloat</td>
<td>The components are unsigned floating-point numbers (used by packed, shared exponent, and some compressed formats)</td>
</tr>
<tr>
<td>SFLOAT</td>
<td>OpTypeFloat</td>
<td>The components are signed floating-point numbers</td>
</tr>
<tr>
<td>SRGB</td>
<td>OpTypeFloat</td>
<td>The R, G, and B components are unsigned normalized values that represent values using sRGB nonlinear encoding, while the A component (if one exists) is a regular unsigned normalized value</td>
</tr>
</tbody>
</table>

\(n\) is the number of bits in the component.

The suffix \_PACK\(nn\) indicates that the format is packed into an underlying type with \(nn\) bits. The suffix \_mPACK\(nn\) is a short-hand that indicates that the format has several components (which may or may not be stored in separate planes) that are each packed into an underlying type with \(nn\) bits.

The suffix \_BLOCK\ indicates that the format is a block-compressed format, with the representation of multiple pixels encoded interdependently within a region.

Table 57. Interpretation of Compression Scheme

<table>
<thead>
<tr>
<th>Compression scheme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>Block Compression. See Block-Compressed Image Formats.</td>
</tr>
<tr>
<td>ETC2</td>
<td>Ericsson Texture Compression. See ETC Compressed Image Formats.</td>
</tr>
<tr>
<td>EAC</td>
<td>ETC2 Alpha Compression. See ETC Compressed Image Formats.</td>
</tr>
<tr>
<td>ASTC</td>
<td>Adaptive Scalable Texture Compression (LDR Profile). See ASTC Compressed Image Formats.</td>
</tr>
</tbody>
</table>

For multi-planar images, the components in separate planes are separated by underscores, and the number of planes is indicated by the addition of a _2PLANE or _3PLANE suffix. Similarly, the separate
aspects of depth-stencil formats are separated by underscores, although these are not considered separate planes. Formats are suffixed by _422 to indicate that planes other than the first are reduced in size by a factor of two horizontally or that the R and B values appear at half the horizontal frequency of the G values, _420 to indicate that planes other than the first are reduced in size by a factor of two both horizontally and vertically, and _444 for consistency to indicate that all three planes of a three-planar image are the same size.

Note
No common format has a single plane containing both R and B channels but does not store these channels at reduced horizontal resolution.

43.1.4. Representation and Texel Block Size

Color formats must be represented in memory in exactly the form indicated by the format’s name. This means that promoting one format to another with more bits per component and/or additional components must not occur for color formats. Depth/stencil formats have more relaxed requirements as discussed below.

Each format has a texel block size, the number of bytes used to store one texel block (a single addressable element of an uncompressed image, or a single compressed block of a compressed image). The texel block size for each format is shown in the Compatible formats table.

The representation of non-packed formats is that the first component specified in the name of the format is in the lowest memory addresses and the last component specified is in the highest memory addresses. See Byte mappings for non-packed/compressed color formats. The in-memory ordering of bytes within a component is determined by the host endianness.

Table 58. Byte mappings for non-packed/compressed color formats

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>← Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8_*</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8G8_*</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8G8B8_*</td>
</tr>
<tr>
<td>B</td>
<td>G</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8_*</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R8G8B8A8_*</td>
</tr>
<tr>
<td>B</td>
<td>G</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8A8_*</td>
</tr>
<tr>
<td>G₀</td>
<td>B</td>
<td>G₁</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_G8B8G8R8_422_UNORM</td>
</tr>
<tr>
<td>B</td>
<td>G₀</td>
<td>R</td>
<td>G₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16_*</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16616_*</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16616B16_*</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VK_FORMAT_R16616B16A16_*</td>
</tr>
</tbody>
</table>
Packed formats store multiple components within one underlying type. The bit representation is that the first component specified in the name of the format is in the most-significant bits and the last component specified is in the least-significant bits of the underlying type. The in-memory ordering of bytes comprising the underlying type is determined by the host endianness.

Table 59. Bit mappings for packed 8-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_R4G4_UNORM_PACK8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>

Table 60. Bit mappings for packed 16-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_R4G4B4A4_UNORM_PACK16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>G</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_B4G4R4A4_UNORM_PACK16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>G</td>
</tr>
<tr>
<td>R</td>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>G</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>G</td>
<td>B</td>
</tr>
</tbody>
</table>
Table 61. Bit mappings for packed 32-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_R5G6B5_UNORM_PACK16</th>
<th>VK_FORMAT_B5G6R5_UNORM_PACK16</th>
<th>VK_FORMAT_R5G5B5A1_UNORM_PACK16</th>
<th>VK_FORMAT_B5G5R5A1_UNORM_PACK16</th>
<th>VK_FORMAT_A1R5G5B5_UNORM_PACK16</th>
<th>VK_FORMAT_R10X6_UNORM_PACK16</th>
<th>VK_FORMAT_R12X4_UNORM_PACK16</th>
<th>VK_FORMAT_B10G11R11_UFLOAT_PACK32</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3 2 1 0 3 2 1 0 3 2 1 0 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0 4 3 2 1 0</td>
<td>4 3 2 1 0 5 4 3 2 1 0 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0 4 3 2 1 0</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 61. Bit mappings for packed 32-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>VK_FORMAT_ABB8G8R8_*_PACK32</th>
<th>VK_FORMAT_A2R10G10B10_*_PACK32</th>
<th>VK_FORMAT_A2B10G10R10_*_PACK32</th>
<th>VK_FORMAT_B10G11R11_UFLOAT_PACK32</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0</td>
<td>1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0</td>
<td>1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 43. Formats | 2297
43.1.5. Depth/Stencil Formats

Depth/stencil formats are considered opaque and need not be stored in the exact number of bits per texel or component ordering indicated by the format enum. However, implementations must not substitute a different depth or stencil precision than that described in the format (e.g. D16 must not be implemented as D24 or D32).

43.1.6. Format Compatibility Classes

Uncompressed color formats are compatible with each other if they occupy the same number of bits per texel block. Compressed color formats are compatible with each other if the only difference between them is the numerical type of the uncompressed pixels (e.g. signed vs. unsigned, or SRGB vs. UNORM encoding). Each depth/stencil format is only compatible with itself. In the following table, all the formats in the same row are compatible.

Table 62. Compatible Formats

<table>
<thead>
<tr>
<th>Class, Texel Block Size, # Texels/Block</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>VK_FORMAT_R4G4_UNORM_PACK8,</td>
</tr>
<tr>
<td>Block size 1 byte</td>
<td>VK_FORMAT_R8_UNORM,</td>
</tr>
<tr>
<td>1 texel/block</td>
<td>VK_FORMAT_R8_SNORM,</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8_USCALED,</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8_SSCALED,</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8_UINT,</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8_UINT,</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8_SINT,</td>
</tr>
<tr>
<td></td>
<td>VK_FORMAT_R8_SRGB</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>16-bit</td>
<td>VK_FORMAT_R4G4B4A4_UNORM_PACK16, VK_FORMAT_B4G4R4A4_UNORM_PACK16, VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT, VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT, VK_FORMAT_R5G6B5_UNORM_PACK16, VK_FORMAT_B5G6R5_UNORM_PACK16, VK_FORMAT_R5G5B5A1_UNORM_PACK16, VK_FORMAT_B5G5R5A1_UNORM_PACK16, VK_FORMAT_A1R5G5B5_UNORM_PACK16, VK_FORMAT_R8G8_UNORM, VK_FORMAT_R8G8_SNORM, VK_FORMAT_R8G8_USCALED, VK_FORMAT_R8G8_SScaled, VK_FORMAT_R8G8_UINT, VK_FORMAT_R8G8_SINT, VK_FORMAT_R8G8_SRGB, VK_FORMAT_R16_UNORM, VK_FORMAT_R16_SNORM, VK_FORMAT_R16_USCALED, VK_FORMAT_R16_SScaled, VK_FORMAT_R16_UINT, VK_FORMAT_R16_SINT, VK_FORMAT_R16_SFLOAT, VK_FORMAT_R10X6_UNORM_PACK16, VK_FORMAT_R12X4_UNORM_PACK16</td>
</tr>
<tr>
<td>24-bit</td>
<td>VK_FORMAT_R8G8B8_UNORM, VK_FORMAT_R8G8B8_SNORM, VK_FORMAT_R8G8B8_USCALED, VK_FORMAT_R8G8B8_SScaled, VK_FORMAT_R8G8B8_UINT, VK_FORMAT_R8G8B8_SINT, VK_FORMAT_R8G8B8_SRGB, VK_FORMAT_B8G8R8_UNORM, VK_FORMAT_B8G8R8_SNORM, VK_FORMAT_B8G8R8_USCALED, VK_FORMAT_B8G8R8_SScaled, VK_FORMAT_B8G8R8_UINT, VK_FORMAT_B8G8R8_SINT, VK_FORMAT_B8G8R8_SRGB</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>32-bit, Block size 4 bytes, 1 texel/block</td>
<td>VK_FORMAT_R8G8B8A8_UNORM, VK_FORMAT_R8G8B8A8_SNORM, VK_FORMAT_R8G8B8A8_USCALED, VK_FORMAT_R8G8B8A8_SSCALED, VK_FORMAT_R8G8B8A8_UINT, VK_FORMAT_R8G8B8A8_SINT, VK_FORMAT_B8G8R8A8_UNORM, VK_FORMAT_B8G8R8A8_SNORM, VK_FORMAT_B8G8R8A8_USCALED, VK_FORMAT_B8G8R8A8_SSCALED, VK_FORMAT_B8G8R8A8_UINT, VK_FORMAT_B8G8R8A8_SINT, VK_FORMAT_A8B8G8R8_UNORM_PACK32, VK_FORMAT_A8B8G8R8_SNORM_PACK32, VK_FORMAT_A8B8G8R8_USCALED_PACK32, VK_FORMAT_A8B8G8R8_SSCALED_PACK32, VK_FORMAT_A8B8G8R8_UINT_PACK32, VK_FORMAT_A8B8G8R8_SINT_PACK32, VK_FORMAT_A10B10G10R10_UNORM_PACK32, VK_FORMAT_A10B10G10R10_SNORM_PACK32, VK_FORMAT_A10B10G10R10_USCALED_PACK32, VK_FORMAT_A10B10G10R10_SSCALED_PACK32, VK_FORMAT_A10B10G10R10_UINT_PACK32, VK_FORMAT_A10B10G10R10_SINT_PACK32, VK_FORMAT_R16G16_UNORM, VK_FORMAT_R16G16_SNORM, VK_FORMAT_R16G16_USCALED, VK_FORMAT_R16G16_SSCALED, VK_FORMAT_R16G16_UINT, VK_FORMAT_R16G16_SINT, VK_FORMAT_R16G16_SFLOAT, VK_FORMAT_R32_UINT, VK_FORMAT_R32_SINT, VK_FORMAT_R32_SFLOAT, VK_FORMAT_B10G11R11_UFLOAT_PACK32, VK_FORMAT_E5B9G9R9_UFLOAT_PACK32, VK_FORMAT_R10X6G10X6_UNORM_2PACK16, VK_FORMAT_R12X4G12X4_UNORM_2PACK16</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>32-bit G8B8G8R8 Block size 4 bytes 1 texel/block</td>
<td>VK_FORMAT_G8B8G8R8_422_UNORM</td>
</tr>
<tr>
<td>32-bit B8G8R8G8 Block size 4 bytes 1 texel/block</td>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
</tr>
<tr>
<td>64-bit R10G10B10A10 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_R10G10B10A10_4PACK16</td>
</tr>
<tr>
<td>64-bit G10B10G10R10 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_G10B10G10R10_4PACK16</td>
</tr>
<tr>
<td>64-bit B10G10R10G10 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_B10G10R10G10_4PACK16</td>
</tr>
<tr>
<td>64-bit R12G12B12A12 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_R12G12B12A12_4PACK16</td>
</tr>
<tr>
<td>64-bit G12B12G12R12 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_G12B12G12R12_4PACK16</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>64-bit B12G12R12G12 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</td>
</tr>
<tr>
<td>64-bit G16B16G16R16 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_G16B16G16R16_422_UNORM</td>
</tr>
<tr>
<td>64-bit B16G16R16G16 Block size 8 bytes 1 texel/block</td>
<td>VK_FORMAT_B16G16R16G16_422_UNORM</td>
</tr>
<tr>
<td>96-bit Block size 12 bytes 1 texel/block</td>
<td>VK_FORMAT_R32G32B32_UINT, VK_FORMAT_R32G32B32_SINT, VK_FORMAT_R32G32B32_SFLOAT</td>
</tr>
<tr>
<td>128-bit Block size 16 bytes 1 texel/block</td>
<td>VK_FORMAT_R32G32B32A32_UINT, VK_FORMAT_R32G32B32A32_SINT, VK_FORMAT_R32G32B32A32_SFLOAT, VK_FORMAT_R64G64_UINT, VK_FORMAT_R64G64_SINT, VK_FORMAT_R64G64_SFLOAT</td>
</tr>
<tr>
<td>192-bit Block size 24 bytes 1 texel/block</td>
<td>VK_FORMAT_R64G64B64_UINT, VK_FORMAT_R64G64B64_SINT, VK_FORMAT_R64G64B64_SFLOAT</td>
</tr>
<tr>
<td>256-bit Block size 32 bytes 1 texel/block</td>
<td>VK_FORMAT_R64G64B64A64_UINT, VK_FORMAT_R64G64B64A64_SINT, VK_FORMAT_R64G64B64A64_SFLOAT</td>
</tr>
<tr>
<td>BC1_RGB (64 bit) Block size 8 bytes 16 texels/block</td>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
</tr>
<tr>
<td>BC1_RGBA (64 bit) Block size 8 bytes 16 texels/block</td>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
</tr>
<tr>
<td>BC2 (128 bit) Block size 16 bytes 16 texels/block</td>
<td>VK_FORMAT_BC2_UNORM_BLOCK, VK_FORMAT_BC2_SRGB_BLOCK</td>
</tr>
<tr>
<td>BC3 (128 bit) Block size 16 bytes 16 texels/block</td>
<td>VK_FORMAT_BC3_UNORM_BLOCK, VK_FORMAT_BC3_SRGB_BLOCK</td>
</tr>
<tr>
<td>BC4 (64 bit) Block size 8 bytes 16 texels/block</td>
<td>VK_FORMAT_BC4_UNORM_BLOCK, VK_FORMAT_BC4_SNORM_BLOCK</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>BC5 (128 bit) 16 texels/block</td>
<td>VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK</td>
</tr>
<tr>
<td>BC6H (128 bit) 16 texels/block</td>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
</tr>
<tr>
<td>BC7 (128 bit) 16 texels/block</td>
<td>VK_FORMAT_BC7_UNORM_BLOCK, VK_FORMAT_BC7_SRGB_BLOCK</td>
</tr>
<tr>
<td>ETC2_RGB (64 bit) 16 texels/block</td>
<td>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</td>
</tr>
<tr>
<td>ETC2_RGBA (64 bit) 16 texels/block</td>
<td>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</td>
</tr>
<tr>
<td>ETC2_EAC_RGBA (64 bit) 16 texels/block</td>
<td>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</td>
</tr>
<tr>
<td>EAC_R (64 bit) 16 texels/block</td>
<td>VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK</td>
</tr>
<tr>
<td>EAC_RG (128 bit) 16 texels/block</td>
<td>VK_FORMAT_EAC_R11G11_UNORM_BLOCK, VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
</tr>
<tr>
<td>ASTC_4x4 (128 bit) 16 texels/block</td>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_5x4 (128 bit) 20 texels/block</td>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_5x5 (128 bit) 25 texels/block</td>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_6x5 (128 bit) 30 texels/block</td>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_6x6 (128 bit) 36 texels/block</td>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>ASTC_8x5 (128 bit) Block size 16 bytes 40 texels/block</td>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_8x6 (128 bit) Block size 16 bytes 48 texels/block</td>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_8x8 (128 bit) Block size 16 bytes 64 texels/block</td>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x5 (128 bit) Block size 16 bytes 50 texels/block</td>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x6 (128 bit) Block size 16 bytes 60 texels/block</td>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK, VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x8 (128 bit) Block size 16 bytes 80 texels/block</td>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK, VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_10x10 (128 bit) Block size 16 bytes 100 texels/block</td>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK, VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_12x10 (128 bit) Block size 16 bytes 120 texels/block</td>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK, VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_12x12 (128 bit) Block size 16 bytes 144 texels/block</td>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK, VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT, VK_FORMAT_ASTC_12x12_SRGB_BLOCK</td>
</tr>
<tr>
<td>D16 (16 bit) Block size 2 bytes 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM</td>
</tr>
<tr>
<td>D24 (32 bit) Block size 4 bytes 1 texel/block</td>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
</tr>
<tr>
<td>D32 (32 bit) Block size 4 bytes 1 texel/block</td>
<td>VK_FORMAT_D32_SFLOAT</td>
</tr>
<tr>
<td>S8 (8 bit) Block size 1 byte 1 texel/block</td>
<td>VK_FORMAT_S8_UINT</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>D16S8 (24 bit) Block size 3 bytes 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
</tr>
<tr>
<td>D24S8 (32 bit) Block size 4 bytes 1 texel/block</td>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
</tr>
<tr>
<td>D32S8 (40 bit) Block size 5 bytes 1 texel/block</td>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
</tr>
<tr>
<td>8-bit 3-plane 420 Block size (1,1,1) bytes 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</td>
</tr>
<tr>
<td>8-bit 2-plane 420 Block size (1,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_2PLANE_420_UNORM</td>
</tr>
<tr>
<td>8-bit 3-plane 422 Block size (1,1,1) bytes 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</td>
</tr>
<tr>
<td>8-bit 2-plane 422 Block size (1,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_2PLANE_422_UNORM</td>
</tr>
<tr>
<td>8-bit 3-plane 444 Block size (1,1,1) bytes 1 texel/block</td>
<td>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</td>
</tr>
<tr>
<td>10-bit 3-plane 420 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 2-plane 420 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 422 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 2-plane 422 Block size (2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>10-bit 3-plane 444 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>12-bit 3-plane 420 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 420 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 422 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 2-plane 422 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
</tr>
<tr>
<td>12-bit 3-plane 444 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
</tr>
<tr>
<td>16-bit 3-plane 420 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
</tr>
<tr>
<td>16-bit 2-plane 420 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</td>
</tr>
<tr>
<td>16-bit 3-plane 422 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
</tr>
<tr>
<td>16-bit 2-plane 422 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</td>
</tr>
<tr>
<td>16-bit 3-plane 444 Block size (2,2,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</td>
</tr>
<tr>
<td>8-bit 2-plane 444 Block size (1,2) bytes 1 texel/block</td>
<td>VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT</td>
</tr>
<tr>
<td>10-bit 2-plane 444 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT</td>
</tr>
<tr>
<td>12-bit 2-plane 444 Block size (2,4) bytes 1 texel/block</td>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT</td>
</tr>
</tbody>
</table>
43.2. Format Properties

To query supported format features which are properties of the physical device, call:

```c
// Provided by VK_VERSION_1_0
void vkGetPhysicalDeviceFormatProperties(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties* pFormatProperties);
```

- `physicalDevice` is the physical device from which to query the format properties.
- `format` is the format whose properties are queried.
- `pFormatProperties` is a pointer to a `VkFormatProperties` structure in which physical device properties for `format` are returned.

### Valid Usage (Implicit)

- `VUID-vkGetPhysicalDeviceFormatProperties-physicalDevice-parameter`  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `VUID-vkGetPhysicalDeviceFormatProperties-format-parameter`  
  `format` must be a valid `VkFormat` value
- `VUID-vkGetPhysicalDeviceFormatProperties-pFormatProperties-parameter`  
  `pFormatProperties` must be a valid pointer to a `VkFormatProperties` structure

The `VkFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkFormatProperties {
    VkFormatFeatureFlags linearTilingFeatures;
    VkFormatFeatureFlags optimalTilingFeatures;
    VkFormatFeatureFlags bufferFeatures;
} VkFormatProperties;
```

- `linearTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_LINEAR`.
- `optimalTilingFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by images created with a `tiling` parameter of `VK_IMAGE_TILING_OPTIMAL`.  

<table>
<thead>
<tr>
<th>Class, Texel Block Size, # Texels/Block</th>
<th>Formats</th>
</tr>
</thead>
</table>
| 16-bit 2-plane 444  
Block size (2,4) bytes  
1 texel/block          | VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT |
• `bufferFeatures` is a bitmask of `VkFormatFeatureFlagBits` specifying features supported by buffers.

**Note**
If no format feature flags are supported, the format itself is not supported, and images of that format cannot be created.

If `format` is a block-compressed format, then `bufferFeatures` must not support any features for the format.

If `format` is not a multi-plane format then `linearTilingFeatures` and `optimalTilingFeatures` must not contain `VK_FORMAT_FEATURE_DISJOINT_BIT`.

Bits which can be set in the `VkFormatProperties` features `linearTilingFeatures`, `optimalTilingFeatures`, `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`, and `bufferFeatures` are:

```c
// Provided by VK_VERSION_1_0
typedef enum VkFormatFeatureFlagBits {
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT = 0x00000001,
    VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT = 0x00000002,
    VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT = 0x00000004,
    VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000008,
    VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT = 0x00000010,
    VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT = 0x00000020,
    VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT = 0x00000040,
    VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT = 0x00000080,
    VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT = 0x00000100,
    VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000200,
    VK_FORMAT_FEATURE_BLIT_SRC_BIT = 0x00000400,
    VK_FORMAT_FEATURE_BLIT_DST_BIT = 0x00000800,
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT = 0x00001000,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_TRANSFER_SRC_BIT = 0x00004000,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_TRANSFER_DST_BIT = 0x00008000,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT = 0x00020000,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT = 0x00040000,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT = 0x00080000,
    // Provided by VK_VERSION_1_1
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT = 0x00100000
} VkFormatFeatureFlagBits;
```
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT = 0x00200000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_DISJOINT_BIT = 0x00400000,
// Provided by VK_VERSION_1_1
VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT = 0x00800000,
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT = 0x00010000,
// Provided by VK_IMG_filter_cubic
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG = 0x00002000,
#if defined VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_decode_queue
VK_FORMAT_FEATURE_VIDEO_DECODE_OUTPUT_BIT_KHR = 0x02000000,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_decode_queue
VK_FORMAT_FEATURE_VIDEO_DECODE_DPB_BIT_KHR = 0x04000000,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_encode_queue
VK_FORMAT_FEATURE_VIDEO_ENCODE_INPUT_BIT_KHR = 0x08000000,
#endif
#if defined VK_ENABLE_BETA_EXTENSIONS
// Provided by VK_KHR_video_encode_queue
VK_FORMAT_FEATURE_VIDEO_ENCODE_DPB_BIT_KHR = 0x10000000,
#endif
// Provided by VK_KHR_acceleration_structure
VK_FORMAT_FEATURE_ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR = 0x20000000,
// Provided by VK_EXT_fragment_density_map
VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT = 0x01000000,
// Provided by VK_KHR_fragment_shading_rate
VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR = 0x40000000,
// Provided by VK_KHR_maintenance1
VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR = VK_FORMAT_FEATURE_TRANSFER_SRC_BIT,
// Provided by VK_KHR_maintenance1
VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR = VK_FORMAT_FEATURE_TRANSFER_DST_BIT,
// Provided by VK_EXT_sampler_filter_minmax
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT = VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT,
// Provided by VK_KHR_sampler_yCBCR_conversion
VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT_KHR = VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT,
// Provided by VK_KHR_sampler_yCBCR_conversion
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT_KHR = VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT,
// Provided by VK_KHR_sampler_yCBCR_conversion
VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT_KHR = VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT,
The following bits **may** be set in `linearTilingFeatures`, `optimalTilingFeatures`, and `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`, specifying that the features are supported by images or image views or sampler YCbCr conversion objects created with the queried `vkGetPhysicalDeviceFormatProperties::format`:

- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** specifies that an image view **can** be sampled from.
- **VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT** specifies that an image view **can** be used as a storage image.
- **VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT** specifies that an image view **can** be used as storage image that supports atomic operations.
- **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT** specifies that an image view **can** be used as a framebuffer color attachment and as an input attachment.
- **VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT** specifies that an image view **can** be used as a framebuffer color attachment that supports blending and as an input attachment.
- **VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT** specifies that an image view **can** be used as a framebuffer depth/stencil attachment and as an input attachment.
- **VK_FORMAT_FEATURE_BLIT_SRC_BIT** specifies that an image **can** be used as `srcImage` for the `vkCmdBlitImage2KHR` and `vkCmdBlitImage` commands.
- **VK_FORMAT_FEATURE_BLIT_DST_BIT** specifies that an image **can** be used as `dstImage` for the `vkCmdBlitImage2KHR` and `vkCmdBlitImage` commands.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT** specifies that if **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** is also set, an image view **can** be used with a sampler that has either of `magFilter` or `minFilter` set to `VK_FILTER_LINEAR`, or `mipmapMode` set to `VK_SAMPLER_MIPMAP_MODE_LINEAR`. If **VK_FORMAT_FEATURE_BLIT_SRC_BIT** is also set, an image can be used as the `srcImage` to `vkCmdBlitImage2KHR` and `vkCmdBlitImage` with a filter of `VK_FILTER_LINEAR`. This bit **must** only be exposed for formats that also support the **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT** or **VK_FORMAT_FEATURE_BLIT_SRC_BIT**.
If the format being queried is a depth/stencil format, this bit only specifies that the depth aspect (not the stencil aspect) of an image of this format supports linear filtering, and that linear filtering of the depth aspect is supported whether depth compare is enabled in the sampler or not. If this bit is not present, linear filtering with depth compare disabled is unsupported and linear filtering with depth compare enabled is supported, but may compute the filtered value in an implementation-dependent manner which differs from the normal rules of linear filtering. The resulting value must be in the range [0,1] and should be proportional to, or a weighted average of, the number of comparison passes or failures.

- **VK_FORMAT_FEATURE_TRANSFER_SRC_BIT** specifies that an image can be used as a source image for copy commands.
- **VK_FORMAT_FEATURE_TRANSFER_DST_BIT** specifies that an image can be used as a destination image for copy commands and clear commands.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT** specifies that a VkImage can be used as a sampled image with a min or max VkSamplerReductionMode. This bit must only be exposed for formats that also support the **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT**.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT** specifies that a VkImage can be used with a sampler that has either of magFilter or minFilter set to VK_FILTER_CUBIC_EXT, or be the source image for a blit with filter set to VK_FILTER_CUBIC_EXT. This bit must only be exposed for formats that also support the **VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT**. If the format being queried is a depth/stencil format, this only specifies that the depth aspect is cubic filterable.
- **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y’CbCr conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo xChromaOffset and/or yChromaOffset of VK_CHROMA_LOCATION_MIDPOINT. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_COSITED_EVEN. If a format does not incorporate chroma downsampling (it is not a “422” or “420” format) but the implementation supports sampler Y’CbCr conversion for this format, the implementation must set **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT**.
- **VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT** specifies that an application can define a sampler Y’CbCr conversion using this format as a source, and that an image of this format can be used with a VkSamplerYcbcrConversionCreateInfo xChromaOffset and/or yChromaOffset of VK_CHROMA_LOCATION_COSITED_EVEN. Otherwise both xChromaOffset and yChromaOffset must be VK_CHROMA_LOCATION_MIDPOINT. If neither **VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT** nor **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT** is set, the application must not define a sampler Y’CbCr conversion using this format as a source.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT** specifies that the format can do linear sampler filtering (min/magFilter) whilst sampler Y’CbCr conversion is enabled.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT** specifies that the format can have different chroma, min, and mag filters.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT** specifies that reconstruction is explicit, as described in Chroma Reconstruction. If this bit is not present, reconstruction is implicit by default.
- **VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT** specifies that reconstruction can be forcibly made explicit by setting
 VkSamplerYcbcrConversionCreateInfo::forceExplicitReconstruction to VK_TRUE. If the format being queried supports VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT it must also support VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT.

- VK_FORMAT_FEATURE_DISJOINT_BIT specifies that a multi-planar image can have the VK_IMAGE_CREATE_DISJOINT_BIT set during image creation. An implementation must not set VK_FORMAT_FEATURE_DISJOINT_BIT for single-plane formats.

- VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT specifies that an image view can be used as a fragment density map attachment.

- VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR specifies that an image view can be used as a fragment shading rate attachment. An implementation must not set this feature for formats with numeric type other than *UINT, or set it as a buffer feature.

- VK_FORMAT_FEATURE_VIDEO_DECODE_OUTPUT_BIT_KHR specifies that an image view with this format can be used as an output for video decode operations

- VK_FORMAT_FEATURE_VIDEO_DECODE_DPB_BIT_KHR specifies that an image view with this format can be used as a DPB for video decode operations

- VK_FORMAT_FEATURE_VIDEO_ENCODE_INPUT_BIT_KHR specifies that an image view with this format can be used as an input to video encode operations

- VK_FORMAT_FEATURE_VIDEO_ENCODE_DPB_BIT_KHR specifies that an image view with this format can be used as a DPB for video encode operations

The following bits may be set in bufferFeatures, specifying that the features are supported by buffers or buffer views created with the queried vkGetPhysicalDeviceFormatProperties::format:

- VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT specifies that the format can be used to create a buffer view that can be bound to a VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER descriptor.

- VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT specifies that the format can be used to create a buffer view that can be bound to a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor.

- VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT specifies that atomic operations are supported on VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER with this format.

- VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT specifies that the format can be used as a vertex attribute format (VkVertexInputAttributeDescription::format).

- VK_FORMAT_FEATURE_ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR specifies that the format can be used as the vertex format when creating an acceleration structure (VkAccelerationStructureGeometryTrianglesDataKHR::vertexFormat). This format can also be used as the vertex format in host memory when doing host acceleration structure builds.

```c
// Provided by VK_VERSION_1_0
typedef VkFlags VkFormatFeatureFlags;

VkFormatFeatureFlags is a bitmask type for setting a mask of zero or more VkFormatFeatureFlagBits.
```
To query supported format features which are properties of the physical device, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceFormatProperties2(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties2* pFormatProperties);
```

or the equivalent command

```c
// Provided by VK_KHR_get_physical_deviceProperties2
void vkGetPhysicalDeviceFormatProperties2KHR(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkFormatProperties2* pFormatProperties);
```

- `physicalDevice` is the physical device from which to query the format properties.
- `format` is the format whose properties are queried.
- `pFormatProperties` is a pointer to a `VkFormatProperties2` structure in which physical device properties for `format` are returned.

`vkGetPhysicalDeviceFormatProperties2` behaves similarly to `vkGetPhysicalDeviceFormatProperties`, with the ability to return extended information in a `pNext` chain of output structures.

### Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceFormatProperties2-physicalDevice-parameter**
  `physicalDevice` must be a valid `VkPhysicalDevice` handle
- **VUID-vkGetPhysicalDeviceFormatProperties2-format-parameter**
  `format` must be a valid `VkFormat` value
- **VUID-vkGetPhysicalDeviceFormatProperties2-pFormatProperties-parameter**
  `pFormatProperties` must be a valid pointer to a `VkFormatProperties2` structure

The `VkFormatProperties2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkFormatProperties formatProperties;
} VkFormatProperties2;
```

or the equivalent
typedef VkFormatProperties2 VkFormatProperties2KHR;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **formatProperties** is a VkFormatProperties structure describing features supported by the requested format.

### Valid Usage (Implicit)

- **VUID-VkFormatProperties2-sType-sType**
  - The **sType** value of each struct in the **pNext** chain must be unique.
- **VUID-VkFormatProperties2-pNext-pNext**
  - Each **pNext** member of any structure (including this one) in the **pNext** chain must be either NULL or a pointer to a valid instance of VkDrmFormatModifierPropertiesListEXT, VkVideoProfileKHR, or VkVideoProfilesKHR.
- **VUID-VkFormatProperties2-sType-unique**
  - Each **pNext** member of any structure (including this one) in the **pNext** chain must be either NULL or a pointer to a valid instance of VkDrmFormatModifierPropertiesListEXT, VkVideoProfileKHR, or VkVideoProfilesKHR.

To obtain the list of Linux DRM format modifiers compatible with a VkFormat, add a VkDrmFormatModifierPropertiesListEXT structure to the **pNext** chain of VkFormatProperties2.

The VkDrmFormatModifierPropertiesListEXT structure is defined as:

```c
typedef struct VkDrmFormatModifierPropertiesListEXT {
    VkStructureType                      sType;
    void*                                pNext;
    uint32_t                             drmFormatModifierCount;
    VkDrmFormatModifierPropertiesEXT*    pDrmFormatModifierProperties;
} VkDrmFormatModifierPropertiesListEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **drmFormatModifierCount** is an inout parameter related to the number of modifiers compatible with the format, as described below.
- **pDrmFormatModifierProperties** is either NULL or an array of VkDrmFormatModifierPropertiesEXT structures.

If **pDrmFormatModifierProperties** is NULL, then the function returns in **drmFormatModifierCount** the number of modifiers compatible with the queried format. Otherwise, the application must set **drmFormatModifierCount** to the length of the array **pDrmFormatModifierProperties**; the function will write at most **drmFormatModifierCount** elements to the array, and will return in
drmFormatModifierCount the number of elements written.

Among the elements in array pDrmFormatModifierProperties, each returned drmFormatModifier must be unique.

Valid Usage (Implicit)

• VUID-VkDrmFormatModifierPropertiesListEXT-sType-sType
  sType must be VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT

The VkDrmFormatModifierPropertiesEXT structure describes properties of a VkFormat when that format is combined with a Linux DRM format modifier. These properties, like those of VkFormatProperties2, are independent of any particular image.

The VkDrmFormatModifierPropertiesEXT structure is defined as:

```
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkDrmFormatModifierPropertiesEXT {
  uint64_t    drmFormatModifier;
  uint32_t    drmFormatModifierPlaneCount;
  VkFormatFeatureFlags drmFormatModifierTilingFeatures;
} VkDrmFormatModifierPropertiesEXT;
```

• drmFormatModifier is a Linux DRM format modifier.

• drmFormatModifierPlaneCount is the number of memory planes in any image created with format and drmFormatModifier. An image's memory planecount is distinct from its format planecount, as explained below.

• drmFormatModifierTilingFeatures is a bitmask of VkFormatFeatureFlagBits that are supported by any image created with format and drmFormatModifier.

The returned drmFormatModifierTilingFeatures must contain at least one bit.

The implementation must not return DRM_FORMAT_MOD_INVALID in drmFormatModifier.

An image's memory planecount (as returned by drmFormatModifierPlaneCount) is distinct from its format planecount (in the sense of multi-planar Y'CbCr formats). In VkImageAspectFlags, each VK_IMAGE_ASPECT_MEMORY_PLANEi_BIT_EXT represents a memory plane and each VK_IMAGE_ASPECT_PLANEi_BIT a format plane.

An image’s set of format planes is an ordered partition of the image’s content into separable groups of format channels. The ordered partition is encoded in the name of each VkFormat. For example, VK_FORMAT_G8_B8R8_2PLANE_420_UNORM contains two format planes; the first plane contains the green channel and the second plane contains the blue channel and red channel. If the format name does not contain PLANE, then the format contains a single plane; for example, VK_FORMAT_R8G8B8A8_UNORM. Some commands, such as vkCmdCopyBufferToImage, do not operate on all format channels in the image, but instead operate only on the format planes explicitly chosen by the application and operate on each format plane independently.
An image’s set of memory planes is an ordered partition of the image’s memory rather than the image’s content. Each memory plane is a contiguous range of memory. The union of an image's memory planes is not necessarily contiguous.

If an image is linear, then the partition is the same for memory planes and for format planes. Therefore, if the returned drmFormatModifier is DRM_FORMAT_MOD_LINEAR, then drmFormatModifierPlaneCount must equal the format planecount, and drmFormatModifierTilingFeatures must be identical to the VkFormatProperties2::linearTilingFeatures returned in the same pNext chain.

If an image is non-linear, then the partition of the image's memory into memory planes is implementation-specific and may be unrelated to the partition of the image's content into format planes. For example, consider an image whose format is VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM, tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, whose drmFormatModifier is not DRM_FORMAT_MOD_LINEAR, and flags lacks VK_IMAGE_CREATE_DISJOINT_BIT. The image has 3 format planes, and commands such as vkCmdCopyBufferToImage act on each format plane independently as if the data of each format plane were separable from the data of the other planes. In a straightforward implementation, the implementation may store the image's content in 3 adjacent memory planes where each memory plane corresponds exactly to a format plane. However, the implementation may also store the image's content in a single memory plane where all format channels are combined using an implementation-private block-compressed format; or the implementation may store the image's content in a collection of 7 adjacent memory planes using an implementation-private sharding technique. Because the image is non-linear and non-disjoint, the implementation has much freedom when choosing the image's placement in memory.

The memory planecount applies to function parameters and structures only when the API specifies an explicit requirement on drmFormatModifierPlaneCount. In all other cases, the memory planecount is ignored.

43.2.1. Potential Format Features

Some valid usage conditions depend on the format features supported by an VkImage whose VkImageTiling is unknown. In such cases the exact VkFormatFeatureFlagBits supported by the VkImage cannot be determined, so the valid usage conditions are expressed in terms of the potential format features of the VkImage format.

The potential format features of a VkFormat are defined as follows:

- The union of VkFormatFeatureFlagBits supported when the VkImageTiling is VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, or VK_IMAGE_TILING_LINEAR if VkFormat is not VK_FORMAT_UNDEFINED

- VkAndroidHardwareBufferFormatPropertiesANDROID::formatFeatures of a valid external format if VkFormat is VK_FORMAT_UNDEFINED

43.3. Required Format Support

Implementations must support at least the following set of features on the listed formats. For images, these features must be supported for every VkImageType (including arrayed and cube
variants) unless otherwise noted. These features are supported on existing formats without needing to advertise an extension or needing to explicitly enable them. Support for additional functionality beyond the requirements listed here is queried using the `vkGetPhysicalDeviceFormatProperties` command.

**Note**

Unless otherwise excluded below, the required formats are supported for all `VkImageCreateFlags` values as long as those flag values are otherwise allowed.

The following tables show which feature bits must be supported for each format. Formats that are required to support `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT` must also support `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT` and `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`.

**Table 63. Key for format feature tables**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>This feature must be supported on the named format</td>
</tr>
<tr>
<td>†</td>
<td>This feature must be supported on at least some of the named formats, with more information in the table where the symbol appears</td>
</tr>
<tr>
<td>‡</td>
<td>This feature must be supported with some caveats or preconditions, with more information in the table where the symbol appears</td>
</tr>
</tbody>
</table>

**Table 64. Feature bits in `optimalTilingFeatures`**

- `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT`
- `VK_FORMAT_FEATURE_TRANSFER_DST_BIT`
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT`
- `VK_FORMAT_FEATURE_BLIT_SRC_BIT`
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`
- `VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT`
- `VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT`
- `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`
- `VK_FORMAT_FEATURE_BLIT_DST_BIT`
- `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`
- `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`
- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT`
Table 66. Mandatory format support: sub-byte channels

<table>
<thead>
<tr>
<th>Format Features</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_UNDEFINED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R4G4_UNORM_PACK8</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R4G4B4A4_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td>VK_FORMAT_B4G4R4A4_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R5G6B5_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>VK_FORMAT_B5G6R5_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_R5G5B5A1_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_A1R5G5B5_UNORM_PACK16</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td>VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT</td>
</tr>
</tbody>
</table>

Format features marked † must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the VkPhysicalDevice4444FormatsFeaturesEXT::formatA4R4G4B4 feature.

Format features marked ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the VkPhysicalDevice4444FormatsFeaturesEXT::formatA4B4G4R4 feature.
Table 67. Mandatory format support: 1-3 byte-sized channels

<table>
<thead>
<tr>
<th>Feature</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_R8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R8_SNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_R8_USCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
<td>VK_FORMAT_R8_SSCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
<td>VK_FORMAT_R8_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_R8_SRGB</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_R8G8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</td>
<td>VK_FORMAT_R8G8_SNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</td>
<td>VK_FORMAT_R8G8_USCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>VK_FORMAT_R8G8_SSCALED</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
<td>VK_FORMAT_R8G8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
<td>VK_FORMAT_R8G8_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
<td>VK_FORMAT_R8G8_SRGB</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
<td>VK_FORMAT_B8G8R8_UNORM</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_B8G8R8_SNORM</td>
</tr>
</tbody>
</table>

Chapter 43. Formats | 2319
<table>
<thead>
<tr>
<th>VK_FORMAT_B8G8R8_USCALED</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_B8G8R8_SSCALED</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_UINT</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_SINT</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_SRGB</td>
</tr>
</tbody>
</table>

Format features marked with ‡ must be supported for `optimalTilingFeatures` if the `VkPhysicalDevice` supports the `shaderStorageImageExtendedFormats` feature.
Table 68. Mandatory format support: 4 byte-sized channels

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</th>
<th>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_DST_BIT</th>
<th>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</th>
<th>VK_FORMAT_FEATURE_BLIT_SRC_BIT</th>
<th>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_SNORM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_USCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_SSCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_UINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_SINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_SRGB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_UNORM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_SNORM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_USCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8A8_SSCALED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_SRGB</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_UNORM_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_SNORM_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_USCALED_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_SSCALED_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_UINT_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_SINT_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A8B8G8R8_SRGB_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Table 69. Mandatory format support: 10- and 12-bit channels

| Format | VK_FORMAT_A2R10G10B10_UNORM_PACK32 | VK_FORMAT_A2R10G10B10_SNORM_PACK32 | VK_FORMAT_A2R10G10B10_USCALED_PACK32 | VK_FORMAT_A2R10G10B10_SSCALED_PACK32 | VK_FORMAT_A2R10G10B10_UINT_PACK32 | VK_FORMAT_A2R10G10B10_SINT_PACK32 | VK_FORMAT_A2B10G10R10_UNORM_PACK32 | VK_FORMAT_A2B10G10R10_SNORM_PACK32 | VK_FORMAT_A2B10G10R10_USCALED_PACK32 | VK_FORMAT_A2B10G10R10_SSCALED_PACK32 | VK_FORMAT_A2B10G10R10_UINT_PACK32 | VK_FORMAT_A2B10G10R10_SINT_PACK32 | VK_FORMAT_R10X6_UNORM_PACK16 | VK_FORMAT_R10X6G10X6_UNORM_2PACK16 | VK_FORMAT_R12X4_UNORM_PACK16 | VK_FORMAT_R12X4G12X4_UNORM_2PACK16 |
|--------|--------------------------------|---------------------------------|---------------------------------|---------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|--------------------------------|
|        | ✓                             | ✓                               | ‡                               | ✓                               | ✓                             | ✓                             | ✓                               | ✓                               | ✓                               | ‡                               | ✓                             | ✓                             | ✓                               | ✓                               | ‡                               | ✓                               |

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
### Table 70. Mandatory format support: 16-bit channels

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‡</td>
<td>‡</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Chapter 43. Formats | 2323
| Format features marked with ‡ | must be supported for optimal Tiling Features if the
| VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature. |
### Table 71. Mandatory format support: 32-bit channels

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

If the shaderImageFloat32Atomics or the shaderImageFloat32AtomicAdd feature is supported, VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT and VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT must be advertised in optimalTilingFeatures for VK_FORMAT_R32_SFLOAT.
Table 72. Mandatory format support: 64-bit/uneven channels

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_R64_UINT</th>
<th>VK_FORMAT_R64_SINT</th>
<th>VK_FORMAT_R64_SFLOAT</th>
<th>VK_FORMAT_R64G64_UINT</th>
<th>VK_FORMAT_R64G64_SINT</th>
<th>VK_FORMAT_R64G64_SFLOAT</th>
<th>VK_FORMAT_R64G64B64_UINT</th>
<th>VK_FORMAT_R64G64B64_SINT</th>
<th>VK_FORMAT_R64G64B64_SFLOAT</th>
<th>VK_FORMAT_R64G64B64A64_UINT</th>
<th>VK_FORMAT_R64G64B64A64_SINT</th>
<th>VK_FORMAT_R64G64B64A64_SFLOAT</th>
<th>VK_FORMAT_B10G11R11_UFLOAT_PACK32</th>
<th>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</th>
</tr>
</thead>
</table>
| Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.

If the shaderImageInt64Atomics feature is supported, VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT and VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT must be advertised in optimalTilingFeatures for both VK_FORMAT_R64_UINT and VK_FORMAT_R64_SINT.
Table 73. Mandatory format support: depth/stencil with \texttt{VkImageType VK\_IMAGE\_TYPE\_2D}

<table>
<thead>
<tr>
<th>Format</th>
<th>FE1</th>
<th>FE2</th>
<th>FE3</th>
<th>FE4</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{VK_FORMAT_D16_UNORM}</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_X8_D24_UNORM_PACK32}</td>
<td>✓</td>
<td></td>
<td></td>
<td>†</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_D32_SFLOAT}</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>†</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_S8_UINT}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_D16_UNORM_S8_UINT}</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_D24_UNORM_S8_UINT}</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_D32_SFLOAT_S8_UINT}</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
</tbody>
</table>

\texttt{VK\_FORMAT\_FEATURE\_DEPTH\_STENCIL\_ATTACHMENT\_BIT} feature \textbf{must} be supported for at least one of \texttt{VK\_FORMAT\_X8\_D24\_UNORM\_PACK32} and \texttt{VK\_FORMAT\_D32\_SFLOAT}, and \textbf{must} be supported for at least one of \texttt{VK\_FORMAT\_D24\_UNORM\_S8\_UINT} and \texttt{VK\_FORMAT\_D32\_SFLOAT\_S8\_UINT}.
### Table 74. Mandatory format support: BC compressed formats with VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D

<table>
<thead>
<tr>
<th>Format</th>
<th>Feature Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT, VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT, VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT, VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT, VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT, VK_FORMAT_FEATURE_BLIT_DST_BIT, VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT, VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT, VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT, VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT, VK_FORMAT_FEATURE_BLIT_SRC_BIT, VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC2_SRGB_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC3_SRGB_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_Block</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC7_SRGB_BLOCK</td>
<td></td>
</tr>
</tbody>
</table>

The VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT, VK_FORMAT_FEATURE_BLIT_SRC_BIT and VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT features must be supported in optimalTilingFeatures for all the formats in at least one of: this table, Mandatory format support: ETC2 and EAC compressed formats with VkImageType VK_IMAGE_TYPE_2D, or Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D.
Table 75. Mandatory format support: ETC2 and EAC compressed formats with \texttt{VkImageType}

<table>
<thead>
<tr>
<th>Format</th>
<th>Ongoing Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11_SNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11G11_UNORM_BLOCK}</td>
<td>† † †</td>
</tr>
<tr>
<td>\texttt{VK_FORMAT_EAC_R11G11_SNORM_BLOCK}</td>
<td>† † †</td>
</tr>
</tbody>
</table>

The \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT}, \texttt{VK_FORMAT_FEATURE_BLIT_SRC_BIT} and \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT} features must be supported in \texttt{optimalTilingFeatures} for all the formats in at least one of: this table, Mandatory format support: BC compressed formats with \texttt{VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D}, or Mandatory format support: ASTC LDR compressed formats with \texttt{VkImageType VK_IMAGE_TYPE_2D}.
Table 76. Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D

<table>
<thead>
<tr>
<th>Format</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
</tbody>
</table>
VK_FORMAT_ASTC_10x8_SRGB_BLOCK  † † †
VK_FORMAT_ASTC_10x10_UNORM_BLOCK  † † †
VK_FORMAT_ASTC_10x10_SRGB_BLOCK  † † †
VK_FORMAT_ASTC_12x10_UNORM_BLOCK  † † †
VK_FORMAT_ASTC_12x10_SRGB_BLOCK  † † †
VK_FORMAT_ASTC_12x12_UNORM_BLOCK  † † †
VK_FORMAT_ASTC_12x12_SRGB_BLOCK  † † †

The VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT, VK_FORMAT_FEATURE_BLIT_SRC_BIT and VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT features must be supported in optimalTilingFeatures for all the formats in at least one of: this table, Mandatory format support: BC compressed formats with VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D, or Mandatory format support: ETC2 and EAC compressed formats with VkImageType VK_IMAGE_TYPE_2D.

If cubic filtering is supported, VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT must be supported for the following image view types:

- VK_IMAGE_VIEW_TYPE_2D
- VK_IMAGE_VIEW_TYPE_2D_ARRAY

for the following formats:

- VK_FORMAT_R4G4_UNORM_PACK8
- VK_FORMAT_R4G4B4A4_UNORM_PACK16
- VK_FORMAT_B4G4R4A4_UNORM_PACK16
- VK_FORMAT_R566B5_UNORM_PACK16
- VK_FORMAT_B565R5_UNORM_PACK16
- VK_FORMAT_R565B5A1_UNORM_PACK16
- VK_FORMAT_B565R5A1_UNORM_PACK16
- VK_FORMAT_A1R565B5_UNORM_PACK16
- VK_FORMAT_R8_UNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R8_SRGB
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R8G8_SRGB
- VK_FORMAT_R8G8B8_UNORM
- VK_FORMAT_R8G8B8_SNORM
- VK_FORMAT_R8G8B8_SRGB
- VK_FORMAT_B8G8R8_UNORM
- VK_FORMAT_B8G8R8_SRGB
• VK_FORMAT_B8G8R8_SNORM
• VK_FORMAT_B8G8R8_SRGB
• VK_FORMAT_R8G8B8A8_UNORM
• VK_FORMAT_R8G8B8A8_SNORM
• VK_FORMAT_R8G8B8A8_SRGB
• VK_FORMAT_B8G8R8A8_UNORM
• VK_FORMAT_B8G8R8A8_SNORM
• VK_FORMAT_B8G8R8A8_SRGB
• VK_FORMAT_A8B8G8R8_UNORM_PACK32
• VK_FORMAT_A8B8G8R8_SNORM_PACK32
• VK_FORMAT_A8B8G8R8_SRGB_PACK32

If ETC compressed formats are supported, **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT** must be supported for the following image view types:

• VK_IMAGE_VIEW_TYPE_2D
• VK_IMAGE_VIEW_TYPE_2D_ARRAY

for the following additional formats:

• VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK
• VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
• VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK
• VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
• VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK
• VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK

If cubic filtering is supported for any other formats, the following image view types **must** be supported for those formats:

• VK_IMAGE_VIEW_TYPE_2D
• VK_IMAGE_VIEW_TYPE_2D_ARRAY

To be used with `VkImageView` with `subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT`, sampler Y’CbCr conversion **must** be enabled for the following formats:

Table 77. Formats requiring sampler Y’CbCr conversion for **VK_IMAGE_ASPECT_COLOR_BIT** image views

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT</td>
<td>Suitable for 2D or 2D array image view types.</td>
</tr>
<tr>
<td>Format</td>
<td>Planes</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>VK_FORMAT_G8B8G8R8_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8G8_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_420_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8R8_2PLANE_422_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_R10X6G10X6R10X6A10X6_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_R12X4G12X4B12X4A12X4_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G16B16G16R16_422_UNORM</td>
<td>1</td>
</tr>
<tr>
<td>VK_FORMAT_B16G16R16G16_422_UNORM</td>
<td>1</td>
</tr>
</tbody>
</table>

Chapter 43. Formats | 2333
<table>
<thead>
<tr>
<th>Format Features</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_420_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_422_UNORM</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM</td>
<td>3</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16_R16_2PLANE_444_UNORM_EXT</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT</td>
<td>2</td>
</tr>
<tr>
<td>VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT</td>
<td>2</td>
</tr>
</tbody>
</table>

Format features marked † must be supported for optimal Tiling Features with VkImageType VK_IMAGE_TYPE_2D if the VkPhysicalDevice supports the VkPhysicalDeviceSamplerYcbcrConversionFeatures feature.

Implementations are not required to support the VK_IMAGE_CREATE_SPARSE_BINDING_BIT, VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, or VK_IMAGE_CREATE_SPARSE_ALIASED_BIT VkImageCreateFlags for the above formats that require sampler Y'CbCr conversion. To determine whether the implementation supports sparse image creation flags with these formats use vkGetPhysicalDeviceImageFormatProperties or vkGetPhysicalDeviceImageFormatProperties2.

VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT must be supported for the following formats if the fragment density map feature is enabled:

- VK_FORMAT_R8G8_UNORM

VK_FORMAT_FEATURE_ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR must be supported for the following formats if the accelerationStructure feature is enabled:

- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32_SFLOAT
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R16G16B16A16_SNORM

VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR must be supported for the following formats if the attachmentFragmentShadingRate feature is supported:

- VK_FORMAT_R8_UINT
Chapter 44. Additional Capabilities

This chapter describes additional capabilities beyond the minimum capabilities described in the Limits and Formats chapters, including:

- Additional Image Capabilities
- Additional Buffer Capabilities
- Optional Semaphore Capabilities
- Optional Fence Capabilities
- Timestamp Calibration Capabilities

44.1. Additional Image Capabilities

Additional image capabilities, such as larger dimensions or additional sample counts for certain image types, or additional capabilities for linear tiling format images, are described in this section.

To query additional capabilities specific to image types, call:

```c
// Provided by VK_VERSION_1_0
VkResult vkGetPhysicalDeviceImageFormatProperties(
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkImageType                                 type,
    VkImageTiling                               tiling,
    VkImageUsageFlags                           usage,
    VkImageCreateFlags                          flags,
    VkImageFormatProperties*                    pImageFormatProperties);
```

- **physicalDevice** is the physical device from which to query the image capabilities.
- **format** is a VkFormat value specifying the image format, corresponding to VkImageCreateInfo::format.
- **type** is a VkImageType value specifying the image type, corresponding to VkImageCreateInfo::imageType.
- **tiling** is a VkImageTiling value specifying the image tiling, corresponding to VkImageCreateInfo::tiling.
- **usage** is a bitmask of VkImageUsageFlagBits specifying the intended usage of the image, corresponding to VkImageCreateInfo::usage.
- **flags** is a bitmask of VkImageCreateFlagBits specifying additional parameters of the image, corresponding to VkImageCreateInfo::flags.
- **pImageFormatProperties** is a pointer to a VkImageFormatProperties structure in which capabilities are returned.

The format, type, tiling, usage, and flags parameters correspond to parameters that would be...
consumed by `vkCreateImage` (as members of `VkImageCreateInfo`).

If `format` is not a supported image format, or if the combination of `format`, `type`, `tiling`, `usage`, and `flags` is not supported for images, then `vkGetPhysicalDeviceImageFormatProperties` returns `VK_ERROR_FORMAT_NOT_SUPPORTED`.

The limitations on an image format that are reported by `vkGetPhysicalDeviceImageFormatProperties` have the following property: if `usage1` and `usage2` of type `VkImageUsageFlags` are such that the bits set in `usage1` are a subset of the bits set in `usage2`, and `flags1` and `flags2` of type `VkImageCreateFlags` are such that the bits set in `flags1` are a subset of the bits set in `flags2`, then the limitations for `usage1` and `flags1` must be no more strict than the limitations for `usage2` and `flags2`, for all values of `format`, `type`, and `tiling`.

### Valid Usage

- VUID-vkGetPhysicalDeviceImageFormatProperties-tiling-02248
  
  tiling must not be `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`. (Use `vkGetPhysicalDeviceImageFormatProperties2` instead)

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceImageFormatProperties-physicalDevice-parameter
  
  physicalDevice must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceImageFormatProperties-format-parameter
  
  format must be a valid `VkFormat` value

- VUID-vkGetPhysicalDeviceImageFormatProperties-type-parameter
  
  type must be a valid `VkImageType` value

- VUID-vkGetPhysicalDeviceImageFormatProperties-tiling-parameter
  
  tiling must be a valid `VkImageTiling` value

- VUID-vkGetPhysicalDeviceImageFormatProperties-usage-parameter
  
  usage must be a valid combination of `VkImageUsageFlagBits` values

- VUID-vkGetPhysicalDeviceImageFormatProperties-usage-requiredbitmap
  
  usage must not be 0

- VUID-vkGetPhysicalDeviceImageFormatProperties-flags-parameter
  
  flags must be a valid combination of `VkImageCreateFlagBits` values

  
  pImageFormatProperties must be a valid pointer to a `VkImageFormatProperties` structure
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_FORMAT_NOT_SUPPORTED

The `VkImageFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_0
typedef struct VkImageFormatProperties {
    VkExtent3D maxExtent;
    uint32_t maxMipLevels;
    uint32_t maxArrayLayers;
    VkSampleCountFlags sampleCounts;
    VkDeviceSize maxResourceSize;
} VkImageFormatProperties;
```

- `maxExtent` are the maximum image dimensions. See the Allowed Extent Values section below for how these values are constrained by type.

- `maxMipLevels` is the maximum number of mipmap levels. `maxMipLevels` must be equal to the number of levels in the complete mipmap chain based on the `maxExtent.width`, `maxExtent.height`, and `maxExtent.depth`, except when one of the following conditions is true, in which case it may instead be 1:
  - `vkGetPhysicalDeviceImageFormatProperties::tiling` was `VK_IMAGE_TILING_LINEAR`
  - `VkPhysicalDeviceImageFormatInfo2::tiling` was `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`
  - the `VkPhysicalDeviceImageFormatInfo2::pNext` chain included a `VkPhysicalDeviceExternalImageFormatInfo` structure with a handle type included in the `handleTypes` member for which mipmap image support is not required
  - image format is one of those listed in Formats requiring sampler Y’C₉C₉ conversion for `VK_IMAGE_ASPECT_COLOR_BIT` image views
  - `flags` contains `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`

- `maxArrayLayers` is the maximum number of array layers. `maxArrayLayers` must be no less than `VkPhysicalDeviceLimits::maxImageArrayLayers`, except when one of the following conditions is true, in which case it may instead be 1:
  - tiling is `VK_IMAGE_TILING_LINEAR`
  - tiling is `VK_IMAGE_TILING_OPTIMAL` and type is `VK_IMAGE_TYPE_3D`
  - format is one of those listed in Formats requiring sampler Y’C₉C₉ conversion for
VK_IMAGE_ASPECT_COLOR_BIT image views

- If tiling is VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT, then maxArrayLayers must not be 0.
- sampleCounts is a bitmask of VkSampleCountFlagBits specifying all the supported sample counts for this image as described below.
- maxResourceSize is an upper bound on the total image size in bytes, inclusive of all image subresources. Implementations may have an address space limit on total size of a resource, which is advertised by this property. maxResourceSize must be at least $2^{31}$.

Note
There is no mechanism to query the size of an image before creating it, to compare that size against maxResourceSize. If an application attempts to create an image that exceeds this limit, the creation will fail and vkCreateImage will return VK_ERROR_OUT_OF_DEVICE_MEMORY. While the advertised limit must be at least $2^{31}$, it may not be possible to create an image that approaches that size, particularly for VK_IMAGE_TYPE_1D.

If the combination of parameters to vkGetPhysicalDeviceImageFormatProperties is not supported by the implementation for use in vkCreateImage, then all members of VkImageFormatProperties will be filled with zero.

Note
Filling VkImageFormatProperties with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility.

To determine the image capabilities compatible with an external memory handle type, call:

```
// Provided by VK_NV_external_memory_capabilities
VkResult vkGetPhysicalDeviceExternalImageFormatPropertiesNV(
    VkPhysicalDevice physicalDevice,
    VkFormat format,
    VkImageType type,
    VkImageTiling tiling,
    VkImageUsageFlags usage,
    VkImageCreateFlags flags,
    VkExternalMemoryHandleTypeFlagsNV externalHandleType,
    VkExternalImageFormatPropertiesNV* pExternalImageFormatProperties);
```

- physicalDevice is the physical device from which to query the image capabilities
- format is the image format, corresponding to VkImageCreateInfo::format.
- type is the image type, corresponding to VkImageCreateInfo::imageType.
- tiling is the image tiling, corresponding to VkImageCreateInfo::tiling.
- usage is the intended usage of the image, corresponding to VkImageCreateInfo::usage.
- flags is a bitmask describing additional parameters of the image, corresponding to
VkImageCreateInfo::flags.

- `externalHandleType` is either one of the bits from `VkExternalMemoryHandleTypeFlagBitsNV`, or 0.

- `pExternalImageFormatProperties` is a pointer to a `VkExternalImageFormatPropertiesNV` structure in which capabilities are returned.

If `externalHandleType` is 0, `pExternalImageFormatProperties->imageFormatProperties` will return the same values as a call to `vkGetPhysicalDeviceImageFormatProperties`, and the other members of `pExternalImageFormatProperties` will all be 0. Otherwise, they are filled in as described for `VkExternalImageFormatPropertiesNV`.

---

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-physicalDevice-parameter
  **physicalDevice** must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-format-parameter
  **format** must be a valid `VkFormat` value

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-type-parameter
  **type** must be a valid `VkImageType` value

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-tiling-parameter
  **tiling** must be a valid `VkImageTiling` value

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-usage-parameter
  **usage** must be a valid combination of `VkImageUsageFlagBits` values

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-usage-requiredbitmask
  **usage** must not be 0

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-flags-parameter
  **flags** must be a valid combination of `VkImageCreateFlagBits` values

- VUID-vkGetPhysicalDeviceExternalImageFormatPropertiesNV-externalHandleType-parameter
  **externalHandleType** must be a valid combination of `VkExternalMemoryHandleTypeFlagBitsNV` values

  **pExternalImageFormatProperties** must be a valid pointer to a `VkExternalImageFormatPropertiesNV` structure
Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
• VK_ERROR_FORMAT_NOT_SUPPORTED

The `VkExternalImageFormatPropertiesNV` structure is defined as:

```c
// Provided by VK_NV_external_memory_capabilities
typedef struct VkExternalImageFormatPropertiesNV {
    VkImageFormatProperties              imageFormatProperties;
    VkExternalMemoryFeatureFlagsNV       externalMemoryFeatures;
    VkExternalMemoryHandleTypeFlagsNV    exportFromImportedHandleTypes;
    VkExternalMemoryHandleTypeFlagsNV    compatibleHandleTypes;
} VkExternalImageFormatPropertiesNV;
```

• `imageFormatProperties` will be filled in as when calling `vkGetPhysicalDeviceImageFormatProperties`, but the values returned may vary depending on the external handle type requested.

• `externalMemoryFeatures` is a bitmask of `VkExternalMemoryFeatureFlagBitsNV`, indicating properties of the external memory handle type (`vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType`) being queried, or 0 if the external memory handle type is 0.

• `exportFromImportedHandleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` containing a bit set for every external handle type that may be used to create memory from which the handles of the type specified in `vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType` can be exported, or 0 if the external memory handle type is 0.

• `compatibleHandleTypes` is a bitmask of `VkExternalMemoryHandleTypeFlagBitsNV` containing a bit set for every external handle type that may be specified simultaneously with the handle type specified by `vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType` when calling `vkAllocateMemory`, or 0 if the external memory handle type is 0. `compatibleHandleTypes` will always contain `vkGetPhysicalDeviceExternalImageFormatPropertiesNV::externalHandleType`.

Bits which can be set in `VkExternalImageFormatPropertiesNV::externalMemoryFeatures`, indicating properties of the external memory handle type, are:
typedef enum VkExternalMemoryFeatureFlagBitsNV {
  VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV = 0x00000001,
  VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_NV = 0x00000002,
  VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV = 0x00000004,
} VkExternalMemoryFeatureFlagBitsNV;

- **VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV** specifies that external memory of the specified type must be created as a dedicated allocation when used in the manner specified.
- **VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_NV** specifies that the implementation supports exporting handles of the specified type.
- **VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV** specifies that the implementation supports importing handles of the specified type.

typedef VkFlags VkExternalMemoryFeatureFlagsNV;

VkExternalMemoryFeatureFlagsNV is a bitmask type for setting a mask of zero or more VkExternalMemoryFeatureFlagBitsNV.

To query additional capabilities specific to image types, call:

```c
// Provided by VK_VERSION_1_1
VkResult vkGetPhysicalDeviceImageFormatProperties2(
  VkPhysicalDevice physicalDevice,
  const VkPhysicalDeviceImageFormatInfo2* pImageFormatInfo,
  VkImageFormatProperties2* pImageFormatProperties);
```

or the equivalent command

```c
// Provided by VK_KHR_get_physical_device_properties2
VkResult vkGetPhysicalDeviceImageFormatProperties2KHR(
  VkPhysicalDevice physicalDevice,
  const VkPhysicalDeviceImageFormatInfo2* pImageFormatInfo,
  VkImageFormatProperties2* pImageFormatProperties);
```

- **physicalDevice** is the physical device from which to query the image capabilities.
- **pImageFormatInfo** is a pointer to a VkPhysicalDeviceImageFormatInfo2 structure describing the parameters that would be consumed by vkCreateImage.
- **pImageFormatProperties** is a pointer to a VkImageFormatProperties2 structure in which capabilities are returned.

vkGetPhysicalDeviceImageFormatProperties2 behaves similarly to vkGetPhysicalDeviceImageFormatProperties, with the ability to return extended information in a
**pNext** chain of output structures.

### Valid Usage

- VUID-vkGetPhysicalDeviceImageFormatProperties2-pNext-01868

  If the **pNext** chain of **pImageFormatProperties** includes a `VkAndroidHardwareBufferUsageANDROID` structure, the **pNext** chain of **pImageFormatInfo** must include a `VkPhysicalDeviceExternalImageFormatInfo` structure with **handleType** set to `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID`.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceImageFormatProperties2-physicalDevice-parameter
  
  **physicalDevice** **must** be a valid `VkPhysicalDevice` handle.

- VUID-vkGetPhysicalDeviceImageFormatProperties2-pImageFormatInfo-parameter
  
  **pImageFormatInfo** **must** be a valid pointer to a valid `VkPhysicalDeviceImageFormatInfo2` structure.

  
  **pImageFormatProperties** **must** be a valid pointer to a `VkImageFormatProperties2` structure.

### Return Codes

**Success**

- VK_SUCCESS

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY
- VK_ERROR_FORMAT_NOT_SUPPORTED

The `VkPhysicalDeviceImageFormatInfo2` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceImageFormatInfo2 {  
    VkStructureType sType;
    const void* pNext;
    VkFormat format;
    VkImageType type;
    VkImageTiling tiling;
    VkImageUsageFlags usage;
    VkImageCreateFlags flags;
} VkPhysicalDeviceImageFormatInfo2;
```
or the equivalent

```c
// Provided by VK_KHR_get_physical_device_properties2
typedef VkPhysicalDeviceImageFormatInfo2 VkPhysicalDeviceImageFormatInfo2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure. The `pNext` chain of `VkPhysicalDeviceImageFormatInfo2` is used to provide additional image parameters to `vkGetPhysicalDeviceImageFormatProperties2`.
- `format` is a `VkFormat` value indicating the image format, corresponding to `VkImageCreateInfo::format`.
- `type` is a `VkImageType` value indicating the image type, corresponding to `VkImageCreateInfo::imageType`.
- `tiling` is a `VkImageTiling` value indicating the image tiling, corresponding to `VkImageCreateInfo::tiling`.
- `usage` is a bitmask of `VkImageUsageFlagBits` indicating the intended usage of the image, corresponding to `VkImageCreateInfo::usage`.
- `flags` is a bitmask of `VkImageCreateFlagBits` indicating additional parameters of the image, corresponding to `VkImageCreateInfo::flags`.

The members of `VkPhysicalDeviceImageFormatInfo2` correspond to the arguments to `vkGetPhysicalDeviceImageFormatProperties`, with `sType` and `pNext` added for extensibility.

---

**Valid Usage**

- VUID-VkPhysicalDeviceImageFormatInfo2-tiling-02249
  
  `tiling` must be `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` if and only if the `pNext` chain includes `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`

- VUID-VkPhysicalDeviceImageFormatInfo2-tiling-02313
  
  If `tiling` is `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` and `flags` contains `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT`, then the `pNext` chain must include a `VkImageFormatListCreateInfo` structure with non-zero `viewFormatCount`
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImageFormatInfo2-sType-sType
  sType must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2`

- VUID-VkPhysicalDeviceImageFormatInfo2-pNext-pNext
  Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkImageFormatListCreateInfo`, `VkImageStencilUsageCreateInfo`, `VkPhysicalDeviceExternalImageFormatInfo`, `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`, or `VkPhysicalDeviceImageViewImageFormatInfoEXT`

- VUID-VkPhysicalDeviceImageFormatInfo2-sType-unique
  The `sType` value of each struct in the `pNext` chain must be unique

- VUID-VkPhysicalDeviceImageFormatInfo2-format-parameter
  format must be a valid `VkFormat` value

- VUID-VkPhysicalDeviceImageFormatInfo2-type-parameter
  type must be a valid `VkImageType` value

- VUID-VkPhysicalDeviceImageFormatInfo2-tiling-parameter
  tiling must be a valid `VkImageTiling` value

- VUID-VkPhysicalDeviceImageFormatInfo2-usage-parameter
  usage must be a valid combination of `VkImageUsageFlagBits` values

- VUID-VkPhysicalDeviceImageFormatInfo2-usage-requiredbitmask
  usage must not be 0

- VUID-VkPhysicalDeviceImageFormatInfo2-flags-parameter
  flags must be a valid combination of `VkImageCreateFlagBits` values

The `VkImageFormatProperties2` structure is defined as:

```
// Provided by VK_VERSION_1_1
typedef struct VkImageFormatProperties2 {
    VkStructureType sType;
    void* pNext;
    VkImageFormatProperties imageFormatProperties;
} VkImageFormatProperties2;
```

or the equivalent

```
// Provided by VK_KHR_get_physical_device_properties2
typedef VkImageFormatProperties2 VkImageFormatProperties2KHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure. The `pNext` chain of `VkImageFormatProperties2` is used to allow the specification of additional capabilities to be
returned from `vkGetPhysicalDeviceImageFormatProperties2`.

- `imageFormatProperties` is a `VkImageFormatProperties` structure in which capabilities are returned.

If the combination of parameters to `vkGetPhysicalDeviceImageFormatProperties2` is not supported by the implementation for use in `vkCreateImage`, then all members of `imageFormatProperties` will be filled with zero.

_Formal_ Filling `imageFormatProperties` with zero for unsupported formats is an exception to the usual rule that output structures have undefined contents on error. This exception was unintentional, but is preserved for backwards compatibility. This exception only applies to `imageFormatProperties`, not `sType`, `pNext`, or any structures chained from `pNext`.

---

**Valid Usage (Implicit)**

- `VUID-VkImageFormatProperties2-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2`

- `VUID-VkImageFormatProperties2-pNext-pNext`
  - Each `pNext` member of any structure (including this one) in the `pNext` chain must be either `NULL` or a pointer to a valid instance of `VkAndroidHardwareBufferUsageANDROID`, `VkExternalImageFormatProperties`, `VkFilterCubicImageViewImageFormatPropertiesEXT`, `VkSamplerYcbcrConversionImageFormatProperties`, or `VkTextureLODGatherFormatPropertiesAMD`

- `VUID-VkImageFormatProperties2-sType-unique`
  - The `sType` value of each struct in the `pNext` chain must be unique

To determine if texture gather functions that take explicit LOD and/or bias argument values can be used with a given image format, add a `VkTextureLODGatherFormatPropertiesAMD` structure to the `pNext` chain of the `VkImageFormatProperties2` structure in a call to `vkGetPhysicalDeviceImageFormatProperties2`.

The `VkTextureLODGatherFormatPropertiesAMD` structure is defined as:

```c
// Provided by VK_AMD_texture_gather_bias_lod
typedef struct VkTextureLODGatherFormatPropertiesAMD {
    VkStructureType   sType;
    void*              pNext;
    VkBool32           supportsTextureGatherLODBiasAMD;
} VkTextureLODGatherFormatPropertiesAMD;
```

- `sType` is the type of this structure.
- `pNext` is `NULL`.

---

Chapter 44. Additional Capabilities | 2345
• supportsTextureGatherLODBiasAMD tells if the image format can be used with texture gather bias/LOD functions, as introduced by the VK_AMD_texture_gather_bias_lod extension. This field is set by the implementation. User-specified value is ignored.

**Valid Usage (Implicit)**

- VUID-VkTextureLODGatherFormatPropertiesAMD-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD

To determine the image capabilities compatible with an external memory handle type, add a VkPhysicalDeviceExternalImageFormatInfo structure to the pNext chain of the VkPhysicalDeviceImageFormatInfo2 structure and a VkExternalImageFormatProperties structure to the pNext chain of the VkImageFormatProperties2 structure.

The VkPhysicalDeviceExternalImageFormatInfo structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalImageFormatInfo {
    VkStructureType                       sType;
    const void*                           pNext;
    VkExternalMemoryHandleTypeFlagBits    handleType;
} VkPhysicalDeviceExternalImageFormatInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_memory_capabilities
typedef VkPhysicalDeviceExternalImageFormatInfoKHR;
VkPhysicalDeviceExternalImageFormatInfo
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **handleType** is a VkExternalMemoryHandleTypeFlagBits value specifying the memory handle type that will be used with the memory associated with the image.

If **handleType** is 0, vkGetPhysicalDeviceImageFormatProperties2 will behave as if VkPhysicalDeviceExternalImageFormatInfo was not present, and VkExternalImageFormatProperties will be ignored.

If **handleType** is not compatible with the **format**, **type**, **tiling**, **usage**, and **flags** specified in VkPhysicalDeviceImageFormatInfo2, then vkGetPhysicalDeviceImageFormatProperties2 returns VK_ERROR_FORMAT_NOT_SUPPORTED.
Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalImageFormatInfo-sType-sType
  sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO

- VUID-VkPhysicalDeviceExternalImageFormatInfo-handleType-parameter
  If handleType is not 0, handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

Possible values of VkPhysicalDeviceExternalImageFormatInfo::handleType, specifying an external memory handle type, are:
typedef enum VkExternalMemoryHandleTypeFlagBits {
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT = 0x00000008,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT = 0x00000010,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT = 0x00000020,
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT = 0x00000040,
    // Provided by VK_EXT_external_memory_dma_buf
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT = 0x00000200,
    // Provided by VK_ANDROID_external_memory_android_hardware_buffer
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID = 0x00000400,
    // Provided by VK_EXT_external_memory_host
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT = 0x00000080,
    // Provided by VK_EXT_external_memory_host
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT = 0x00000100,
    // Provided by VK_FUCHSIA_external_memory
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA = 0x00000800,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT,
    // Provided by VK_KHR_external_memory_capabilities
    VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT_KHR = VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT,
} VkExternalMemoryHandleTypeFlagBits;

or the equivalent

// Provided by VK_KHR_external_memory_capabilities
typedef VkExternalMemoryHandleTypeFlagBits VkExternalMemoryHandleTypeFlagBitsKHR;

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT** specifies a POSIX file descriptor handle that has
only limited valid usage outside of Vulkan and other compatible APIs. It must be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally, it must be transportable over a socket using an `SCM_RIGHTS` control message. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It must be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying memory resource represented by its Vulkan memory object, and will therefore become invalid when all Vulkan memory objects associated with it are destroyed.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT** specifies an NT handle returned by `IDXGIFactory1::CreateSharedHandle` referring to a Direct3D 10 or 11 texture resource. It owns a reference to the memory used by the Direct3D resource.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT** specifies a global share handle returned by `IDXGIFactory::CreateSharedHandle` referring to a Direct3D 10 or 11 texture resource. It does not own a reference to the underlying Direct3D resource, and will therefore become invalid when all Vulkan memory objects and Direct3D resources associated with it are destroyed.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT** specifies an NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 heap resource. It owns a reference to the resources used by the Direct3D heap.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT** specifies an NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 committed resource. It owns a reference to the memory used by the Direct3D resource.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT** specifies a host pointer returned by a host memory allocation command. It does not own a reference to the underlying memory resource, and will therefore become invalid if the host memory is freed.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT** specifies a host pointer to *host mapped foreign memory*. It does not own a reference to the underlying memory resource, and will therefore become invalid if the foreign memory is unmapped or otherwise becomes no longer available.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT** is a file descriptor for a Linux dma_buf. It owns a reference to the underlying memory resource represented by its Vulkan memory object.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID** specifies an `AHardwareBuffer` object defined by the Android NDK. See [Android Hardware Buffers](#) for more details of this handle type.

- **VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA** is a Zircon handle to a virtual memory object.
Some external memory handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

**Table 78. External memory handle types compatibility**

<table>
<thead>
<tr>
<th>Handle type</th>
<th>VkPhysicalDeviceIDProperties::driverUUID</th>
<th>VkPhysicalDeviceIDProperties::deviceUUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_HEAP_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

**Note**

The above table does not restrict the drivers and devices with which `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT` and `VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT` may be shared, as these handle types inherently mean memory that does not come from the same device, as they import memory from the host or a foreign device, respectively.
Note

Even though the above table does not restrict the drivers and devices with which
\texttt{VK\_EXTERNAL\_MEMORY\_HANDLE\_TYPE\_DMA\_BUF\_BIT\_EXT} may be shared, query
mechanisms exist in the Vulkan API that prevent the import of incompatible dma-
bufs (such as \texttt{vkGetMemoryFdPropertiesKHR}) and that prevent incompatible
usage of dma-bufs (such as \texttt{VkPhysicalDeviceExternalBufferInfo} and
\texttt{VkPhysicalDeviceExternalImageFormatInfo}).

```
// Provided by VK\_VERSION\_1\_1
typedef VkFlags VkExternalMemoryHandleTypeFlags;
```

or the equivalent

```
// Provided by VK\_KHR\_external\_memory\_capabilities
typedef VkExternalMemoryHandleTypeFlags VkExternalMemoryHandleTypeFlagsKHR;
```

\texttt{VkExternalMemoryHandleTypeFlags} is a bitmask type for setting a mask of zero or more
\texttt{VkExternalMemoryHandleTypeFlagBits}.

The \texttt{VkExternalImageFormatProperties} structure is defined as:

```
// Provided by VK\_VERSION\_1\_1
typedef struct VkExternalImageFormatProperties {
    VkStructureType               sType;
    void*                         pNext;
    VkExternalMemoryProperties    externalMemoryProperties;
} VkExternalImageFormatProperties;
```

or the equivalent

```
// Provided by VK\_KHR\_external\_memory\_capabilities
typedef VkExternalImageFormatProperties VkExternalImageFormatPropertiesKHR;
```

- \texttt{sType} is the type of this structure.
- \texttt{pNext} is \texttt{NULL} or a pointer to a structure extending this structure.
- \texttt{externalMemoryProperties} is a \texttt{VkExternalMemoryProperties} structure specifying various
capabilities of the external handle type when used with the specified image creation
parameters.
Valid Usage (Implicit)

- VUID-VkExternalImageFormatProperties-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES`

The `VkExternalMemoryProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalMemoryProperties {
    VkExternalMemoryFeatureFlags externalMemoryFeatures;
    VkExternalMemoryHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalMemoryHandleTypeFlags compatibleHandleTypes;
} VkExternalMemoryProperties;
```

or the equivalent

```c
// Provided by VK_KHR_external_memory_capabilities
typedef VkExternalMemoryProperties VkExternalMemoryPropertiesKHR;
```

- **externalMemoryFeatures** is a bitmask of `VkExternalMemoryFeatureFlagBits` specifying the features of `handleType`

- **exportFromImportedHandleTypes** is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying which types of imported handle `handleType` can be exported from.

- **compatibleHandleTypes** is a bitmask of `VkExternalMemoryHandleTypeFlagBits` specifying handle types which can be specified at the same time as `handleType` when creating an image compatible with external memory.

  `compatibleHandleTypes` **must** include at least `handleType`. Inclusion of a handle type in `compatibleHandleTypes` does not imply the values returned in `VkImageFormatProperties2` will be the same when `VkPhysicalDeviceExternalImageFormatInfo::handleType` is set to that type. The application is responsible for querying the capabilities of all handle types intended for concurrent use in a single image and intersecting them to obtain the compatible set of capabilities.

Bits which **may** be set in `VkExternalMemoryProperties::externalMemoryFeatures`, specifying features of an external memory handle type, are:
typedef enum VkExternalMemoryFeatureFlagBits {
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT = 0x00000001,
    VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT = 0x00000002,
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT = 0x00000004,
} VkExternalMemoryFeatureFlagBits;

or the equivalent

typedef VkExternalMemoryFeatureFlagBits VkExternalMemoryFeatureFlagBitsKHR;

• VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT specifies that images or buffers created with the specified parameters and handle type must use the mechanisms defined by VkMemoryDedicatedRequirements and VkMemoryDedicatedAllocateInfo to create (or import) a dedicated allocation for the image or buffer.

• VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT specifies that handles of this type can be exported from Vulkan memory objects.

• VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT specifies that handles of this type can be imported as Vulkan memory objects.

Because their semantics in external APIs roughly align with that of an image or buffer with a dedicated allocation in Vulkan, implementations are required to report VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT for the following external handle types:

• VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT
• VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_KMT_BIT
• VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D12_RESOURCE_BIT
• VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID for images only

Implementations must not report VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT for buffers with external handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID.

Implementations must not report VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT for images or buffers with external handle type VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT, or VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT.
VkExternalMemoryFeatureFlags is a bitmask type for setting a mask of zero or more VkExternalMemoryFeatureFlagBits.

To query the image capabilities that are compatible with a Linux DRM format modifier, set VkPhysicalDeviceImageFormatInfo2::tiling to VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and add a VkPhysicalDeviceImageDrmFormatModifierInfoEXT structure to the pNext chain of VkPhysicalDeviceImageFormatInfo2.

The VkPhysicalDeviceImageDrmFormatModifierInfoEXT structure is defined as:

```c
// Provided by VK_EXT_image_drm_format_modifier
typedef struct VkPhysicalDeviceImageDrmFormatModifierInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    uint64_t           drmFormatModifier;
    VkSharingMode      sharingMode;
    uint32_t           queueFamilyIndexCount;
    const uint32_t*    pQueueFamilyIndices;
} VkPhysicalDeviceImageDrmFormatModifierInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `drmFormatModifier` is the image’s Linux DRM format modifier, corresponding to VkImageDrmFormatModifierExplicitCreateInfoEXT::modifier or to VkImageDrmFormatModifierListCreateInfoEXT::pModifiers.
- `sharingMode` specifies how the image will be accessed by multiple queue families.
- `queueFamilyIndexCount` is the number of entries in the `pQueueFamilyIndices` array.
- `pQueueFamilyIndices` is a list of queue families that will access the image (ignored if `sharingMode` is not VK_SHARING_MODE_CONCURRENT).

If the `drmFormatModifier` is incompatible with the parameters specified in VkPhysicalDeviceImageFormatInfo2 and its pNext chain, then vkGetPhysicalDeviceImageFormatProperties2 returns VK_ERROR_FORMAT_NOT_SUPPORTED. The implementation must support the query of any `drmFormatModifier`, including unknown and invalid modifier values.
Valid Usage

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-02314
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-02315
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, then `queueFamilyIndexCount` must be greater than 1

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-02316
  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than the `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties2` for the `physicalDevice` that was used to create device

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sType-sType
  `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT`

- VUID-VkPhysicalDeviceImageDrmFormatModifierInfoEXT-sharingMode-parameter
  `sharingMode` must be a valid `VkSharingMode` value

To determine the number of combined image samplers required to support a multi-planar format, add `VkSamplerYcbcrConversionImageFormatProperties` to the `pNext` chain of the `VkImageFormatProperties2` structure in a call to `vkGetPhysicalDeviceImageFormatProperties2`.

The `VkSamplerYcbcrConversionImageFormatProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkSamplerYcbcrConversionImageFormatProperties {
    VkStructureType    sType;
    void*              pNext;
    uint32_t           combinedImageSamplerDescriptorCount;
} VkSamplerYcbcrConversionImageFormatProperties;
```

or the equivalent

```c
// Provided by VK_KHR_sampler_ycbcr_conversion
typedef VkSamplerYcbcrConversionImageFormatProperties
VkSamplerYcbcrConversionImageFormatPropertiesKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
• *combinedImageSamplerDescriptorCount* is the number of combined image sampler descriptors that the implementation uses to access the format.

**Valid Usage (Implicit)**

- VUID-VkSamplerYcbcrConversionImageFormatProperties-sType-sType
  
  **sType must be** VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES

*combinedImageSamplerDescriptorCount* is a number between 1 and the number of planes in the format. A descriptor set layout binding with immutable Y'CBCR conversion samplers will have a maximum *combinedImageSamplerDescriptorCount* which is the maximum across all formats supported by its samplers of the *combinedImageSamplerDescriptorCount* for each format. Descriptor sets with that layout will internally use that maximum *combinedImageSamplerDescriptorCount* descriptors for each descriptor in the binding. This expanded number of descriptors will be consumed from the descriptor pool when a descriptor set is allocated, and counts towards the maxDescriptorSetSamplers, maxDescriptorSetSampledImages, maxPerStageDescriptorSamplers, and maxPerStageDescriptorSampledImages limits.

**Note**

All descriptors in a binding use the same maximum *combinedImageSamplerDescriptorCount* descriptors to allow implementations to use a uniform stride for dynamic indexing of the descriptors in the binding.

For example, consider a descriptor set layout binding with two descriptors and immutable samplers for multi-planar formats that have *VkSamplerYcbcrConversionImageFormatProperties::combinedImageSamplerDescriptorCount* values of 2 and 3 respectively. There are two descriptors in the binding and the maximum *combinedImageSamplerDescriptorCount* is 3, so descriptor sets with this layout consume 6 descriptors from the descriptor pool. To create a descriptor pool that allows allocating four descriptor sets with this layout, *descriptorCount* must be at least 24.

To obtain optimal Android hardware buffer usage flags for specific image creation parameters, add a *VkAndroidHardwareBufferUsageANDROID* structure to the *pNext* chain of a *VkImageFormatProperties2* structure passed to *vkGetPhysicalDeviceImageFormatProperties2*. This structure is defined as:

```c
// Provided by VK_ANDROID_external_memory_android_hardware_buffer
typedef struct VkAndroidHardwareBufferUsageANDROID {
    VkStructureType    sType;
    void*              pNext;
    uint64_t           androidHardwareBufferUsage;
} VkAndroidHardwareBufferUsageANDROID;
```

- *sType* is the type of this structure.
- *pNext* is *NULL* or a pointer to a structure extending this structure.
• **androidHardwareBufferUsage** returns the Android hardware buffer usage flags.

The **androidHardwareBufferUsage** field **must** include Android hardware buffer usage flags listed in the *AHardwareBuffer Usage Equivalence* table when the corresponding Vulkan image usage or image creation flags are included in the *usage* or *flags* fields of **VkPhysicalDeviceImageFormatInfo2**. It **must** include at least one GPU usage flag (**AHARDWAREBUFFER_USAGE_GPU_***), even if none of the corresponding Vulkan usages or flags are requested.

**Note**

Requiring at least one GPU usage flag ensures that Android hardware buffer memory will be allocated in a memory pool accessible to the Vulkan implementation, and that specializing the memory layout based on usage flags does not prevent it from being compatible with Vulkan. Implementations **may** avoid unnecessary restrictions caused by this requirement by using vendor usage flags to indicate that only the Vulkan uses indicated in **VkImageFormatProperties2** are required.

---

**Valid Usage (Implicit)**

- **VUID-VkAndroidHardwareBufferUsageANDROID-sType-sType**

  **sType** must be **VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_USAGE_ANDROID**

To determine if cubic filtering can be used with a given image format and a given image view type add a **VkPhysicalDeviceImageViewImageFormatInfoEXT** structure to the *pNext* chain of the **VkPhysicalDeviceImageFormatInfo2** structure, and a **VkFilterCubicImageViewImageFormatPropertiesEXT** structure to the *pNext* chain of the **VkImageFormatProperties2** structure.

The **VkPhysicalDeviceImageViewImageFormatInfoEXT** structure is defined as:

```c
// Provided by VK_EXT_filter_cubic
typedef struct VkPhysicalDeviceImageViewImageFormatInfoEXT {
    VkStructureType sType;
    void* pNext;
    VkImageViewType imageViewType;
} VkPhysicalDeviceImageViewImageFormatInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **imageViewType** is a **VkImageViewType** value specifying the type of the image view.
The `VkFilterCubicImageViewImageFormatPropertiesEXT` structure is defined as:

```c
// Provided by VK_EXT_filter_cubic
typedef struct VkFilterCubicImageViewImageFormatPropertiesEXT {
    VkStructureType sType;
    void* pNext;
    VkBool32 filterCubic;
    VkBool32 filterCubicMinmax;
} VkFilterCubicImageViewImageFormatPropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **filterCubic** tells if image format, image type and image view type can be used with cubic filtering. This field is set by the implementation. User-specified value is ignored.
- **filterCubicMinmax** tells if image format, image type and image view type can be used with cubic filtering and minmax filtering. This field is set by the implementation. User-specified value is ignored.

### 44.1.1. Supported Sample Counts

`vkGetPhysicalDeviceImageFormatProperties` returns a bitmask of `VkSampleCountFlagBits` in
sampleCounts specifying the supported sample counts for the image parameters.

sampleCounts will be set to VK_SAMPLE_COUNT_1_BIT if at least one of the following conditions is true:

- tiling is VK_IMAGE_TILING_LINEAR
- type is not VK_IMAGE_TYPE_2D
- flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT
- Neither the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag nor the VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT flag in VkFormatProperties:optimalTilingFeatures returned by vkGetPhysicalDeviceFormatProperties is set
- VkPhysicalDeviceExternalImageFormatInfo::handleType is an external handle type for which multisampled image support is not required.
- format is one of those listed in Formats requiring sampler Y’C_aC_b conversion for VK_IMAGE_ASPECT_COLOR_BIT image views
- usage contains VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR
- usage contains VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT

Otherwise, the bits set in sampleCounts will be the sample counts supported for the specified values of usage and format. For each bit set in usage, the supported sample counts relate to the limits in VkPhysicalDeviceLimits as follows:

- If usage includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT and format is a floating- or fixed-point color format, a superset of VkPhysicalDeviceLimits::framebufferColorSampleCounts
- If usage includes VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and format includes a depth aspect, a superset of VkPhysicalDeviceLimits::framebufferDepthSampleCounts
- If usage includes VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and format includes a stencil aspect, a superset of VkPhysicalDeviceLimits::framebufferStencilSampleCounts
- If usage includes VK_IMAGE_USAGE_SAMPLED_BIT, and format includes a color aspect, a superset of VkPhysicalDeviceLimits::sampledImageColorSampleCounts
- If usage includes VK_IMAGE_USAGE_SAMPLED_BIT, and format includes a depth aspect, a superset of VkPhysicalDeviceLimits::sampledImageDepthSampleCounts
- If usage includes VK_IMAGE_USAGE_SAMPLED_BIT, and format is an integer format, a superset of VkPhysicalDeviceLimits::sampledImageIntegerSampleCounts
- If usage includes VK_IMAGE_USAGE_STORAGE_BIT, a superset of VkPhysicalDeviceLimits::storageImageSampleCounts

If multiple bits are set in usage, sampleCounts will be the intersection of the per-usage values described above.

If none of the bits described above are set in usage, then there is no corresponding limit in VkPhysicalDeviceLimits. In this case, sampleCounts must include at least VK_SAMPLE_COUNT_1_BIT.
44.1.2. Allowed Extent Values Based On Image Type

Implementations may support extent values larger than the required minimum/maximum values for certain types of images. VkImageFormatProperties::maxExtent for each type is subject to the constraints below.

Note

Implementations must support images with dimensions up to the required minimum/maximum values for all types of images. It follows that the query for additional capabilities must return extent values that are at least as large as the required values.

For VK_IMAGE_TYPE_1D:

- \( \text{maxExtent.width} \geq \text{VkPhysicalDeviceLimits::maxImageDimension1D} \)
- \( \text{maxExtent.height} = 1 \)
- \( \text{maxExtent.depth} = 1 \)

For VK_IMAGE_TYPE_2D when flags does not contain VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT:

- \( \text{maxExtent.width} \geq \text{VkPhysicalDeviceLimits::maxImageDimension2D} \)
- \( \text{maxExtent.height} \geq \text{VkPhysicalDeviceLimits::maxImageDimension2D} \)
- \( \text{maxExtent.depth} = 1 \)

For VK_IMAGE_TYPE_2D when flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT:

- \( \text{maxExtent.width} \geq \text{VkPhysicalDeviceLimits::maxImageDimensionCube} \)
- \( \text{maxExtent.height} \geq \text{VkPhysicalDeviceLimits::maxImageDimensionCube} \)
- \( \text{maxExtent.depth} = 1 \)

For VK_IMAGE_TYPE_3D:

- \( \text{maxExtent.width} \geq \text{VkPhysicalDeviceLimits::maxImageDimension3D} \)
- \( \text{maxExtent.height} \geq \text{VkPhysicalDeviceLimits::maxImageDimension3D} \)
- \( \text{maxExtent.depth} \geq \text{VkPhysicalDeviceLimits::maxImageDimension3D} \)

44.2. Additional Buffer Capabilities

To query the external handle types supported by buffers, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalBufferProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalBufferInfo* pExternalBufferInfo,
    VkExternalBufferProperties* pExternalBufferProperties);
```
or the equivalent command

```c
// Provided by VK_KHR_external_memory_capabilities
void vkGetPhysicalDeviceExternalBufferPropertiesKHR(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalBufferInfo* pExternalBufferInfo,
    VkExternalBufferProperties* pExternalBufferProperties);
```

- `physicalDevice` is the physical device from which to query the buffer capabilities.
- `pExternalBufferInfo` is a pointer to a `VkPhysicalDeviceExternalBufferInfo` structure describing the parameters that would be consumed by `vkCreateBuffer`.
- `pExternalBufferProperties` is a pointer to a `VkExternalBufferProperties` structure in which capabilities are returned.

### Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceExternalBufferProperties-physicalDevice-parameter
  - `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceExternalBufferProperties-pExternalBufferInfo-parameter
  - `pExternalBufferInfo` must be a valid pointer to a valid `VkPhysicalDeviceExternalBufferInfo` structure
- VUID-vkGetPhysicalDeviceExternalBufferProperties-pExternalBufferProperties-parameter
  - `pExternalBufferProperties` must be a valid pointer to a `VkExternalBufferProperties` structure

The `VkPhysicalDeviceExternalBufferInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalBufferInfo {
    VkStructureType sType;
    const void* pNext;
    VkBufferCreateFlags flags;
    VkBufferUsageFlags usage;
    VkExternalMemoryHandleTypeFlagBits handleType;
} VkPhysicalDeviceExternalBufferInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_memory_capabilities
typedef VkPhysicalDeviceExternalBufferInfo VkPhysicalDeviceExternalBufferInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
• flags is a bitmask of VkBufferCreateFlagBits describing additional parameters of the buffer, corresponding to VkBufferCreateInfo::flags.

• usage is a bitmask of VkBufferUsageFlagBits describing the intended usage of the buffer, corresponding to VkBufferCreateInfo::usage.

• handleType is a VkExternalMemoryHandleTypeFlagBits value specifying the memory handle type that will be used with the memory associated with the buffer.

Valid Usage (Implicit)

- VUID-VkPhysicalDeviceExternalBufferInfo-sType-sType sType must be VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO
- VUID-VkPhysicalDeviceExternalBufferInfo-pNext-pNext pNext must be NULL
- VUID-VkPhysicalDeviceExternalBufferInfo-flags-parameter flags must be a valid combination of VkBufferCreateFlagBits values
- VUID-VkPhysicalDeviceExternalBufferInfo-usage-parameter usage must be a valid combination of VkBufferUsageFlagBits values
- VUID-VkPhysicalDeviceExternalBufferInfo-usage-requiredBitmask usage must not be 0
- VUID-VkPhysicalDeviceExternalBufferInfo-handleType-parameter handleType must be a valid VkExternalMemoryHandleTypeFlagBits value

The VkExternalBufferProperties structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalBufferProperties {
    VkStructureType               sType;
    void*                         pNext;
    VkExternalMemoryProperties    externalMemoryProperties;
} VkExternalBufferProperties;
```

or the equivalent

```c
// Provided by VK_KHR_external_memory_capabilities
typedef VkExternalBufferProperties VkExternalBufferPropertiesKHR;
```

• sType is the type of this structure.

• pNext is NULL or a pointer to a structure extending this structure.

• externalMemoryProperties is a VkExternalMemoryProperties structure specifying various capabilities of the external handle type when used with the specified buffer creation parameters.
44.3. Optional Semaphore Capabilities

Semaphores may support import and export of their payload to external handles. To query the external handle types supported by semaphores, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalSemaphoreProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalSemaphoreInfo* pExternalSemaphoreInfo,
    VkExternalSemaphoreProperties* pExternalSemaphoreProperties);
```

or the equivalent command

```c
// Provided by VK_KHR_external_semaphore_capabilities
void vkGetPhysicalDeviceExternalSemaphorePropertiesKHR(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalSemaphoreInfo* pExternalSemaphoreInfo,
    VkExternalSemaphoreProperties* pExternalSemaphoreProperties);
```

- `physicalDevice` is the physical device from which to query the semaphore capabilities.
- `pExternalSemaphoreInfo` is a pointer to a `VkPhysicalDeviceExternalSemaphoreInfo` structure describing the parameters that would be consumed by `vkCreateSemaphore`.
- `pExternalSemaphoreProperties` is a pointer to a `VkExternalSemaphoreProperties` structure in which capabilities are returned.

Valid Usage (Implicit)

- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-physicalDevice-parameter `physicalDevice` must be a valid `VkPhysicalDevice` handle
- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-pExternalSemaphoreInfo-parameter `pExternalSemaphoreInfo` must be a valid pointer to a `VkPhysicalDeviceExternalSemaphoreInfo` structure
- VUID-vkGetPhysicalDeviceExternalSemaphoreProperties-pExternalSemaphoreProperties-parameter `pExternalSemaphoreProperties` must be a valid pointer to a `VkExternalSemaphoreProperties` structure
The `VkPhysicalDeviceExternalSemaphoreInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalSemaphoreInfo {
    VkStructureType                          sType;
    const void*                              pNext;
    VkExternalSemaphoreHandleTypeFlagBits    handleType;
} VkPhysicalDeviceExternalSemaphoreInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_semaphore_capabilities
typedef VkPhysicalDeviceExternalSemaphoreInfo
VkPhysicalDeviceExternalSemaphoreInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to a structure extending this structure.
- `handleType` is a `VkExternalSemaphoreHandleTypeFlagBits` value specifying the external semaphore handle type for which capabilities will be returned.

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO`
- `pNext` must be `NULL` or a pointer to a valid instance of `VkSemaphoreTypeCreateInfo`
- The `sType` value of each struct in the `pNext` chain must be unique
- `handleType` must be a valid `VkExternalSemaphoreHandleTypeFlagBits` value

Bits which may be set in `VkPhysicalDeviceExternalSemaphoreInfo::handleType`, specifying an external semaphore handle type, are:
// Provided by VK_VERSION_1_1

typedef enum VkExternalSemaphoreHandleTypeFlagBits {
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT = 0x00000008,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT = 0x00000010,
    // Provided by VK_FUCHSIA_external_semaphore
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA = 0x00000080,
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE_BIT =
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT,
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT_KHR =
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT,
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR =
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT,
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR =
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT,
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHR =
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT,
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT_KHR =
        VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT,
} VkExternalSemaphoreHandleTypeFlagBits;

or the equivalent

// Provided by VK_KHR_external_semaphore_capabilities
typedef VkExternalSemaphoreHandleTypeFlagBits
    VkExternalSemaphoreHandleTypeFlagBitsKHR;

• **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally, it **must** be transportable over a socket using an `SCM_RIGHTS` control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.

• **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying synchronization primitive represented by its Vulkan semaphore object.

• **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive.
represented by its Vulkan semaphore object, and will therefore become invalid when all Vulkan semaphore objects associated with it are destroyed.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT** specifies an NT handle returned by ID3D12Device::CreateSharedHandle referring to a Direct3D 12 fence, or ID3D11Device5::CreateFence referring to a Direct3D 11 fence. It owns a reference to the underlying synchronization primitive associated with the Direct3D fence.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D11_FENCE_BIT** is an alias of VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT with the same meaning. It is provided for convenience and code clarity when interacting with D3D11 fences.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT** specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence object. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type must accept any type of sync or fence FD supported by the native system they are running on.

- **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA** specifies a handle to a Zircon event object. It can be used with any native API that accepts a Zircon event handle. Zircon event handles are created with ZX_RIGHTS_BASIC and ZX_RIGHTS_SIGNAL rights. Vulkan on Fuchsia uses only the ZX_EVENT_SIGNALING bit when signaling or waiting.

**Note**

Handles of type **VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT** generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation's discretion. Applications should only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.
Some external semaphore handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

<table>
<thead>
<tr>
<th>Handle type</th>
<th>VkPhysicalDeviceIDProperties::driverUUID</th>
<th>VkPhysicalDeviceIDProperties::deviceUUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
<tr>
<td>VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

// Provided by VK_VERSION_1_1

typedef VkFlags VkExternalSemaphoreHandleTypeFlags;

or the equivalent

// Provided by VK_KHR_external_semaphore_capabilities

typedef VkExternalSemaphoreHandleTypeFlags VkExternalSemaphoreHandleTypeFlagsKHR;

VkExternalSemaphoreHandleTypeFlags is a bitmask type for setting a mask of zero or more VkExternalSemaphoreHandleTypeFlagBits.

The VkExternalSemaphoreProperties structure is defined as:

// Provided by VK_VERSION_1_1

typedef struct VK_EXTERNAL_SEMAPHORE_PROPERTIES {  
VkStructureType sType;  
void* pNext;  
VkExternalSemaphoreHandleTypeFlags exportFromImportedHandleTypes;  
VkExternalSemaphoreHandleTypeFlags compatibleHandleTypes;  
VkExternalSemaphoreFeatureFlags externalSemaphoreFeatures;
} VkExternalSemaphoreProperties;

or the equivalent

// Provided by VK_KHR_external_semaphore_capabilities

typedef VkExternalSemaphoreProperties VkExternalSemaphorePropertiesKHR;
• **sType** is the type of this structure

• **pNext** is NULL or a pointer to a structure extending this structure.

• **exportFromImportedHandleTypes** is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying which types of imported handle `handleType` can be exported from.

• **compatibleHandleTypes** is a bitmask of `VkExternalSemaphoreHandleTypeFlagBits` specifying handle types which can be specified at the same time as `handleType` when creating a semaphore.

• **externalSemaphoreFeatures** is a bitmask of `VkExternalSemaphoreFeatureFlagBits` describing the features of `handleType`.

If `handleType` is not supported by the implementation, then `VkExternalSemaphoreProperties::externalSemaphoreFeatures` will be set to zero.

### Valid Usage (Implicit)

- VUID-VkExternalSemaphoreProperties-sType-sType
  
  **sType must be VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES**

- VUID-VkExternalSemaphoreProperties-pNext-pNext
  
  **pNext must be NULL**

Possible values of `VkExternalSemaphoreProperties::externalSemaphoreFeatures`, specifying the features of an external semaphore handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalSemaphoreFeatureFlagBits {
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT = 0x00000002,
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT_KHR =
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT_KHR =
    // Provided by VK_KHR_external_semaphore_capabilities
    VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT_KHR,
    VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT_KHR,
} VkExternalSemaphoreFeatureFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_external_semaphore_capabilities
typedef VkExternalSemaphoreFeatureFlagBits VkExternalSemaphoreFeatureFlagBitsKHR;
```

- **VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT** specifies that handles of this type can be exported from Vulkan semaphore objects.

- **VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT** specifies that handles of this type can be imported as Vulkan semaphore objects.
or the equivalent

```c
// Provided by VK_KHR_external_semaphore_capabilities
typedef VkExternalSemaphoreFeatureFlagsKHR VkExternalSemaphoreFeatureFlags;
```

`VkExternalSemaphoreFeatureFlags` is a bitmask type for setting a mask of zero or more `VkExternalSemaphoreFeatureFlagBits`.

## 44.4. Optional Fence Capabilities

Fences may support import and export of their payload to external handles. To query the external handle types supported by fences, call:

```c
// Provided by VK_VERSION_1_1
void vkGetPhysicalDeviceExternalFenceProperties(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalFenceInfo* pExternalFenceInfo,
    VkExternalFenceProperties* pExternalFenceProperties);
```

or the equivalent command

```c
// Provided by VK_KHR_external_fence_capabilities
void vkGetPhysicalDeviceExternalFencePropertiesKHR(
    VkPhysicalDevice physicalDevice,
    const VkPhysicalDeviceExternalFenceInfo* pExternalFenceInfo,
    VkExternalFenceProperties* pExternalFenceProperties);
```

- `physicalDevice` is the physical device from which to query the fence capabilities.
- `pExternalFenceInfo` is a pointer to a `VkPhysicalDeviceExternalFenceInfo` structure describing the parameters that would be consumed by `vkCreateFence`.
- `pExternalFenceProperties` is a pointer to a `VkExternalFenceProperties` structure in which capabilities are returned.
Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceExternalFenceProperties-physicalDevice-parameter**
  
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- **VUID-vkGetPhysicalDeviceExternalFenceProperties-pExternalFenceInfo-parameter**
  
  `pExternalFenceInfo` must be a valid pointer to a valid `VkPhysicalDeviceExternalFenceInfo` structure

- **VUID-vkGetPhysicalDeviceExternalFenceProperties-pExternalFenceProperties-parameter**
  
  `pExternalFenceProperties` must be a valid pointer to a `VkExternalFenceProperties` structure

The `VkPhysicalDeviceExternalFenceInfo` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkPhysicalDeviceExternalFenceInfo {
    VkStructureType                      sType;
    const void*                          pNext;
    VkExternalFenceHandleTypeFlagBits    handleType;
} VkPhysicalDeviceExternalFenceInfo;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence_capabilities
typedef VkPhysicalDeviceExternalFenceInfo VkPhysicalDeviceExternalFenceInfoKHR;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `handleType` is a `VkExternalFenceHandleTypeFlagBits` value indicating an external fence handle type for which capabilities will be returned.

**Note**

Handles of type `VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT` generated by the implementation may represent either Linux Sync Files or Android Fences at the implementation’s discretion. Applications **should** only use operations defined for both types of file descriptors, unless they know via means external to Vulkan the type of the file descriptor, or are prepared to deal with the system-defined operation failures resulting from using the wrong type.
Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceExternalFenceInfo-sType-sType**
  - `sType` must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO`

- **VUID-VkPhysicalDeviceExternalFenceInfo-pNext-pNext**
  - `pNext` must be `NULL`

- **VUID-VkPhysicalDeviceExternalFenceInfo-handleType-parameter**
  - `handleType` must be a valid `VkExternalFenceHandleTypeFlagBits` value

Bits which **may** be set in `VkPhysicalDeviceExternalFenceInfo::handleType`, and in the `exportFromImportedHandleTypes` and `compatibleHandleTypes` members of `VkExternalFenceProperties`, to indicate external fence handle types, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalFenceHandleTypeFlagBits {
  VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT = 0x00000001,
  VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT = 0x00000002,
  VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT = 0x00000004,
  VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT = 0x00000008,
} VkExternalFenceHandleTypeFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence_capabilities
typedef VkExternalFenceHandleTypeFlagBits VkExternalFenceHandleTypeFlagBitsKHR;
```

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT** specifies a POSIX file descriptor handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally, it **must** be transportable over a socket using an `SCM_RIGHTS` control message. It owns a reference to the underlying synchronization primitive represented by its Vulkan fence object.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT** specifies an NT handle that has only limited valid usage outside of Vulkan and other compatible APIs. It **must** be compatible with the
functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying synchronization primitive represented by its Vulkan fence object.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT** specifies a global share handle that has only limited valid usage outside of Vulkan and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its Vulkan fence object, and will therefore become invalid when all Vulkan fence objects associated with it are destroyed.

- **VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT** specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type **must** accept any type of sync or fence FD supported by the native system they are running on.
Some external fence handle types can only be shared within the same underlying physical device and/or the same driver version, as defined in the following table:

**Table 80. External fence handle types compatibility**

<table>
<thead>
<tr>
<th>Handle type</th>
<th>VkPhysicalDeviceIDProperties::driverUUID</th>
<th>VkPhysicalDeviceIDProperties::deviceUUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT</td>
<td>Must match</td>
<td>Must match</td>
</tr>
<tr>
<td>VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT</td>
<td>No restriction</td>
<td>No restriction</td>
</tr>
</tbody>
</table>

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalFenceHandleTypeFlags;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence_capabilities
typedef VkExternalFenceHandleTypeFlags VkExternalFenceHandleTypeFlagsKHR;
```

`VkExternalFenceHandleTypeFlags` is a bitmask type for setting a mask of zero or more `VkExternalFenceHandleTypeFlagBits`.

The `VkExternalFenceProperties` structure is defined as:

```c
// Provided by VK_VERSION_1_1
typedef struct VkExternalFenceProperties {
    VkStructureType sType;
    void* pNext;
    VkExternalFenceHandleTypeFlags exportFromImportedHandleTypes;
    VkExternalFenceHandleTypeFlags compatibleHandleTypes;
    VkExternalFenceFeatureFlags externalFenceFeatures;
} VkExternalFenceProperties;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence_capabilities
typedef VkExternalFenceProperties VkExternalFencePropertiesKHR;
```

- `exportFromImportedHandleTypes` is a bitmask of `VkExternalFenceHandleTypeFlagBits` indicating which types of imported handle `handleType` can be exported from.
• **compatibleHandleTypes** is a bitmask of VkExternalFenceHandleTypeFlagBits specifying handle types which can be specified at the same time as handleType when creating a fence.

• **externalFenceFeatures** is a bitmask of VkExternalFenceFeatureFlagBits indicating the features of handleType.

If handleType is not supported by the implementation, then VkExternalFenceProperties::externalFenceFeatures will be set to zero.

### Valid Usage (Implicit)

- **VUID-VkExternalFenceProperties-sType-sType**
  
  sType must be VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES

- **VUID-VkExternalFenceProperties-pNext-pNext**
  
  pNext must be NULL

Bits which may be set in VkExternalFenceProperties::externalFenceFeatures, indicating features of a fence external handle type, are:

```c
// Provided by VK_VERSION_1_1
typedef enum VkExternalFenceFeatureFlagBits {
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT = 0x00000001,
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT = 0x00000002,
    // Provided by VK_KHR_external_fence_capabilities
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT_KHR =
    VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT,
    // Provided by VK_KHR_external_fence_capabilities
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT_KHR =
    VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT,
} VkExternalFenceFeatureFlagBits;
```

or the equivalent

```c
// Provided by VK_KHR_external_fence_capabilities
typedef VkExternalFenceFeatureFlagBits VkExternalFenceFeatureFlagBitsKHR;
```

- **VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT** specifies handles of this type can be exported from Vulkan fence objects.

- **VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT** specifies handles of this type can be imported to Vulkan fence objects.

```c
// Provided by VK_VERSION_1_1
typedef VkFlags VkExternalFenceFeatureFlags;
```

or the equivalent
VkExternalFenceFeatureFlags is a bitmask type for setting a mask of zero or more VkExternalFenceFeatureFlagBits.

### 44.5. Timestamp Calibration Capabilities

To query the set of time domains for which a physical device supports timestamp calibration, call:

```c
VkResult vkGetPhysicalDeviceCalibrateableTimeDomainsEXT(
    VkPhysicalDevice physicalDevice,
    uint32_t* pTimeDomainCount,
    VkTimeDomainEXT* pTimeDomains);
```

- **physicalDevice** is the physical device from which to query the set of calibrateable time domains.
- **pTimeDomainCount** is a pointer to an integer related to the number of calibrateable time domains available or queried, as described below.
- **pTimeDomains** is either NULL or a pointer to an array of VkTimeDomainEXT values, indicating the supported calibrateable time domains.

If **pTimeDomains** is NULL, then the number of calibrateable time domains supported for the given **physicalDevice** is returned in **pTimeDomainCount**. Otherwise, **pTimeDomainCount** must point to a variable set by the user to the number of elements in the **pTimeDomains** array, and on return the variable is overwritten with the number of values actually written to **pTimeDomains**. If the value of **pTimeDomainCount** is less than the number of calibrateable time domains supported, at most **pTimeDomainCount** values will be written to **pTimeDomains**, and **VK_INCOMPLETE** will be returned instead of **VK_SUCCESS**, to indicate that not all the available time domains were returned.

#### Valid Usage (Implicit)

- **VUID-vkGetPhysicalDeviceCalibrateableTimeDomainsEXT-physicalDevice-parameter**
  physicalDevice must be a valid VkPhysicalDevice handle

- **VUID-vkGetPhysicalDeviceCalibrateableTimeDomainsEXT-pTimeDomainCount-parameter**
  pTimeDomainCount must be a valid pointer to a uint32_t value

- **VUID-vkGetPhysicalDeviceCalibrateableTimeDomainsEXT-pTimeDomains-parameter**
  If the value referenced by **pTimeDomainCount** is not 0, and **pTimeDomains** is not NULL, **pTimeDomains** must be a valid pointer to an array of **pTimeDomainCount** VkTimeDomainEXT values
Return Codes

Success

• VK_SUCCESS
• VK_INCOMPLETE

Failure

• VK_ERROR_OUT_OF_HOST_MEMORY
• VK_ERROR_OUT_OF_DEVICE_MEMORY
Chapter 45. Debugging

To aid developers in tracking down errors in the application’s use of Vulkan, particularly in combination with an external debugger or profiler, *debugging extensions* may be available.

The `VkObjectType` enumeration defines values, each of which corresponds to a specific Vulkan handle type. These values *can* be used to associate debug information with a particular type of object through one or more extensions.

```c
// Provided by VK_VERSION_1_0
typedef enum VkObjectType {
    VK_OBJECT_TYPE_UNKNOWN = 0,
    VK_OBJECT_TYPE_INSTANCE = 1,
    VK_OBJECT_TYPE_PHYSICAL_DEVICE = 2,
    VK_OBJECT_TYPE_DEVICE = 3,
    VK_OBJECT_TYPE_QUEUE = 4,
    VK_OBJECT_TYPE_SEMAPHORE = 5,
    VK_OBJECT_TYPE_COMMAND_BUFFER = 6,
    VK_OBJECT_TYPE_FENCE = 7,
    VK_OBJECT_TYPE_DEVICE_MEMORY = 8,
    VK_OBJECT_TYPE_BUFFER = 9,
    VK_OBJECT_TYPE_IMAGE = 10,
    VK_OBJECT_TYPE_IMAGE_VIEW = 11,
    VK_OBJECT_TYPE_COLOR_IMAGE = 12,
    VK_OBJECT_TYPE_DEPTH_STENCIL_IMAGE = 13,
    VK_OBJECT_TYPE_SHADER_IMAGE = 14,
    VK_OBJECT_TYPE_SAMPLED_IMAGE = 15,
    VK_OBJECT_TYPE_ALL_IMAGE_TYPES = 16,
    VK_OBJECT_TYPE_UNIFORM_TEXEL_BUFFER = 17,
    VK_OBJECT_TYPE_STORAGE_TEXEL_BUFFER = 18,
    VK_OBJECT_TYPE_UNIFORM_BUFFER = 19,
    VK_OBJECT_TYPE_STORAGE_BUFFER = 20,
    VK_OBJECT_TYPE_UNIFORM_BUFFER_DYNAMIC = 21,
    VK_OBJECT_TYPE_STORAGE_BUFFER_DYNAMIC = 22,
    VK_OBJECT_TYPE_SHADER_MODULE = 23,
    VK_OBJECT_TYPE_PIPELINE_CACHE = 24,
    VK_OBJECT_TYPE_PIPELINE_LAYOUT = 25,
    VK_OBJECT_TYPE_RENDER_PASS = 26,
    VK_OBJECT_TYPE_PIPELINE = 27,
    VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT = 28,
    VK_OBJECT_TYPE_SAMPLER = 29,
    VK_OBJECT_TYPE_DESCRIPTOR_POOL = 30,
    VK_OBJECT_TYPE_DESCRIPTOR_SET = 31,
    VK_OBJECT_TYPE_FRAMEBUFFER = 32,
    VK_OBJECT_TYPE_COMMAND_POOL = 33,
    VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION = 1000156000,
    VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE = 1000085000,
} VkObjectType;
```
Table 81. VkObjectType and Vulkan Handle Relationship

<table>
<thead>
<tr>
<th>VkObjectType</th>
<th>Vulkan Handle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_OBJECT_TYPE_UNKNOWN</td>
<td>Unknown/Undefined Handle</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_INSTANCE</td>
<td>VkInstance</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PHYSICAL_DEVICE</td>
<td>VkPhysicalDevice</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEVICE</td>
<td>VkDevice</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_QUEUE</td>
<td>VkQueue</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SEMAPHORE</td>
<td>VkSemaphore</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_COMMAND_BUFFER</td>
<td>VkCommandBuffer</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_FENCE</td>
<td>VkFence</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEVICE_MEMORY</td>
<td>VkDeviceMemory</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_BUFFER</td>
<td>VkBuffer</td>
</tr>
</tbody>
</table>

```c
#ifndef VK_ENABLE_BETA_EXTENSIONS
  // Provided by VK_KHR_video_queue
  VK_OBJECT_TYPE_VIDEO_SESSION_KHR = 1000023000,
#endif

#ifdef VK_Enable_BETA_EXTENSIONS
  // Provided by VK_KHR_video_queue
  VK_OBJECT_TYPE_VIDEO_SESSION_PARAMETERS_KHR = 1000023001,
#endif

// Provided by VK_EXT_debug_utils
  VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT = 1000128000,
// Provided by VK_KHR_acceleration_structure
  VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_KHR = 1000150000,
// Provided by VK_EXT_validation_cache
  VK_OBJECT_TYPE_VALIDATION_CACHE_EXT = 1000160000,
// Provided by VK_KHR_acceleration_structure
  VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_NV = 1000165000,
// Provided by VK_INTEL_performance_query
  VK_OBJECT_TYPE_PERFORMANCE_CONFIGURATION_INTEL = 1000210000,
// Provided by VK_KHR_deferred_host_operations
  VK_OBJECT_TYPE_DEFERRED_OPERATION_KHR = 1000268000,
// Provided by VK_NV_device_generated_commands
  VK_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NV = 1000277000,
// Provided by VK_EXT_private_data
  VK_OBJECT_TYPE_PRIVATE_DATA_SLOT_EXT = 1000295000,
// Provided by VK_KHR_descriptor_update_template
  VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR = VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE,
// Provided by VK_KHR_sampler_ycbcr_conversion
  VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_KHR = VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION,
```
<table>
<thead>
<tr>
<th>VkObjectType</th>
<th>Vulkan Handle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_OBJECT_TYPE_IMAGE</td>
<td>VkImage</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_EVENT</td>
<td>VkEvent</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_QUERY_POOL</td>
<td>VkQueryPool</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_BUFFER_VIEW</td>
<td>VkBufferView</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_IMAGE_VIEW</td>
<td>VkImageView</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SHADER_MODULE</td>
<td>VkShaderModule</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PIPELINE_CACHE</td>
<td>VkPipelineCache</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PIPELINE_LAYOUT</td>
<td>VkPipelineLayout</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_RENDER_PASS</td>
<td>VkRenderPass</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PIPELINE</td>
<td>VkPipeline</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT</td>
<td>VkDescriptorSetLayout</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SAMPLER</td>
<td>VkSampler</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_POOL</td>
<td>VkDescriptorPool</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_SET</td>
<td>VkDescriptorSet</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_FRAMEBUFFER</td>
<td>VkFramebuffer</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_COMMAND_POOL</td>
<td>VkCommandPool</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION</td>
<td>VkSamplerYcbcrConversion</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE</td>
<td>VkDescriptorUpdateTemplate</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SURFACE_KHR</td>
<td>VkSurfaceKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_SWAPCHAIN_KHR</td>
<td>VkSwapchainKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DISPLAY_KHR</td>
<td>VkDisplayKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DISPLAY_MODE_KHR</td>
<td>VkDisplayModeKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT</td>
<td>VkDebugReportCallbackEXT</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NV</td>
<td>VkIndirectCommandsLayoutNV</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT</td>
<td>VkDebugUtilsMessageEXT</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_VALIDATION_CACHE_EXT</td>
<td>VkValidationCacheEXT</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_NV</td>
<td>VkAccelerationStructureNV</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_KHR</td>
<td>VkAccelerationStructureKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PERFORMANCE_CONFIGURATION_INTEL</td>
<td>VkPerformanceConfigurationINTEL</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_DEFERRED_OPERATION_KHR</td>
<td>VkDeferredOperationKHR</td>
</tr>
<tr>
<td>VK_OBJECT_TYPE_PRIVATE_DATA_SLOT_EXT</td>
<td>VkPrivateDataSlotEXT</td>
</tr>
</tbody>
</table>

If this Specification was generated with any such extensions included, they will be described in the
45.1. Debug Utilities

Vulkan provides flexible debugging utilities for debugging an application.

The **Object Debug Annotation** section describes how to associate either a name or binary data with a specific Vulkan object.

The **Queue Labels** section describes how to annotate and group the work submitted to a queue.

The **Command Buffer Labels** section describes how to associate logical elements of the scene with commands in a `VkCommandBuffer`.

The **Debug Messengers** section describes how to create debug messenger objects associated with an application supplied callback to capture debug messages from a variety of Vulkan components.

### 45.1.1. Object Debug Annotation

It can be useful for an application to provide its own content relative to a specific Vulkan object. The following commands allow application developers to associate user-defined information with Vulkan objects.

**Object Naming**

An object can be provided a user-defined name by calling `vkSetDebugUtilsObjectNameEXT` as defined below.

```c
// Provided by VK_EXT_debug_utils
VkResult vkSetDebugUtilsObjectNameEXT(
    VkDevice                                    device,
    const VkDebugUtilsObjectNameInfoEXT*        pNameInfo);
```

- `device` is the device that created the object.
- `pNameInfo` is a pointer to a `VkDebugUtilsObjectNameInfoEXT` structure specifying parameters of the name to set on the object.

**Valid Usage**

- `VUID-vkSetDebugUtilsObjectNameEXT-pNameInfo-02587` 
  `pNameInfo->objectType must not be VK_OBJECT_TYPE_UNKNOWN`
- `VUID-vkSetDebugUtilsObjectNameEXT-pNameInfo-02588` 
  `pNameInfo->objectHandle must not be VK_NULL_HANDLE`
Valid Usage (Implicit)

- VUID-vkSetDebugUtilsObjectNameEXT-device-parameter
  
  device must be a valid VkDevice handle

- VUID-vkSetDebugUtilsObjectNameEXT-pNameInfo-parameter
  
  pNameInfo must be a valid pointer to a valid VkDebugUtilsObjectNameInfoEXT structure

Host Synchronization

- Host access to pNameInfo->objectHandle must be externally synchronized

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDebugUtilsObjectNameInfoEXT structure is defined as:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsObjectNameInfoEXT {
  VkStructureType sType;
  const void* pNext;
  VkObjectType objectType;
  uint64_t objectHandle;
  const char* pObjectName;
} VkDebugUtilsObjectNameInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **objectType** is a VkObjectType specifying the type of the object to be named.
- **objectHandle** is the object to be named.
- **pObjectName** is either **NULL** or a null-terminated UTF-8 string specifying the name to apply to objectHandle.

Applications may change the name associated with an object simply by calling vkSetDebugUtilsObjectNameEXT again with a new string. If pObjectName is either **NULL** or an empty string, then any previously set name is removed.
Valid Usage

- **VUID-VkDebugUtilsObjectNameInfoEXT-objectType-02589**
  
  If `objectType` is `VK_OBJECT_TYPE_UNKNOWN`, `objectHandle` must not be `VK_NULL_HANDLE`.

- **VUID-VkDebugUtilsObjectNameInfoEXT-objectType-02590**
  
  If `objectType` is not `VK_OBJECT_TYPE_UNKNOWN`, `objectHandle` must be `VK_NULL_HANDLE` or a valid Vulkan handle of the type associated with `objectType` as defined in the `VkObjectType` and Vulkan Handle Relationship table.

Valid Usage (Implicit)

- **VUID-VkDebugUtilsObjectNameInfoEXT-sType-sType**
  
  `sType` must be `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT`.

- **VUID-VkDebugUtilsObjectNameInfoEXT-pNext-pNext**
  
  `pNext` must be `NULL`.

- **VUID-VkDebugUtilsObjectNameInfoEXT-objectType-parameter**
  
  `objectType` must be a valid `VkObjectType` value.

- **VUID-VkDebugUtilsObjectNameInfoEXT-pObjectName-parameter**
  
  If `pObjectName` is not `NULL`, `pObjectName` must be a null-terminated UTF-8 string.

Object Data Association

In addition to setting a name for an object, debugging and validation layers may have uses for additional binary data on a per-object basis that have no other place in the Vulkan API.

For example, a `VkShaderModule` could have additional debugging data attached to it to aid in offline shader tracing.

Additional data can be attached to an object by calling `vkSetDebugUtilsObjectTagEXT` as defined below.

```c
// Provided by VK_EXT_debug_utils
VkResult vkSetDebugUtilsObjectTagEXT(
    VkDevice                                    device,
    const VkDebugUtilsObjectTagInfoEXT*         pTagInfo);
```

- `device` is the device that created the object.

- `pTagInfo` is a pointer to a `VkDebugUtilsObjectTagInfoEXT` structure specifying parameters of the tag to attach to the object.
Valid Usage (Implicit)

- VUID-vkSetDebugUtilsObjectTagEXT-device-parameter
  device must be a valid VkDevice handle

- VUID-vkSetDebugUtilsObjectTagEXT-pTagInfo-parameter
  pTagInfo must be a valid pointer to a valid VkDebugUtilsObjectTagInfoEXT structure

Host Synchronization

- Host access to pTagInfo->objectHandle must be externally synchronized

Return Codes

Success

- VK_SUCCESS

Failure

- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDebugUtilsObjectTagInfoEXT structure is defined as:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsObjectTagInfoEXT {
    VkStructureType   sType;
    const void*       pNext;
    VkObjectType      objectType;
    uint64_t           objectHandle;
    uint64_t           tagName;
    size_t             tagSize;
    const void*       pTag;
} VkDebugUtilsObjectTagInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **objectType** is a VkObjectType specifying the type of the object to be named.
- **objectHandle** is the object to be tagged.
- **tagName** is a numerical identifier of the tag.
- **tagSize** is the number of bytes of data to attach to the object.
- **pTag** is a pointer to an array of tagSize bytes containing the data to be associated with the object.
The `tagName` parameter gives a name or identifier to the type of data being tagged. This can be used by debugging layers to easily filter for only data that can be used by that implementation.

### Valid Usage

- **VUID-VkDebugUtilsObjectTagInfoEXT-objectType-01908**
  - `objectType` **must** not be `VK_OBJECT_TYPE_UNKNOWN`

- **VUID-VkDebugUtilsObjectTagInfoEXT-objectHandle-01910**
  - `objectHandle` **must** be a valid Vulkan handle of the type associated with `objectType` as defined in the `VkObjectType` and `Vulkan Handle Relationship` table

### Valid Usage (Implicit)

- **VUID-VkDebugUtilsObjectTagInfoEXT-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT`

- **VUID-VkDebugUtilsObjectTagInfoEXT-pNext-pNext**
  - `pNext` **must** be `NULL`

- **VUID-VkDebugUtilsObjectTagInfoEXT-objectType-parameter**
  - `objectType` **must** be a valid `VkObjectType` value

- **VUID-VkDebugUtilsObjectTagInfoEXT-pTag-parameter**
  - `pTag` **must** be a valid pointer to an array of `tagSize` bytes

- **VUID-VkDebugUtilsObjectTagInfoEXT-tagSize-arraylength**
  - `tagSize` **must** be greater than `0`

### 45.1.2. Queue Labels

All Vulkan work must be submitted using queues. It is possible for an application to use multiple queues, each containing multiple command buffers, when performing work. It can be useful to identify which queue, or even where in a queue, something has occurred.

To begin identifying a region using a debug label inside a queue, you may use the `vkQueueBeginDebugUtilsLabelEXT` command.

Then, when the region of interest has passed, you may end the label region using `vkQueueEndDebugUtilsLabelEXT`.

Additionally, a single debug label may be inserted at any time using `vkQueueInsertDebugUtilsLabelEXT`.

A queue debug label region is opened by calling:
// Provided by VK_EXT_debug_utils
void vkQueueBeginDebugUtilsLabelEXT(
    VkQueue queue,
    const VkDebugUtilsLabelEXT* pLabelInfo);

- queue is the queue in which to start a debug label region.
- pLabelInfo is a pointer to a VkDebugUtilsLabelEXT structure specifying parameters of the label region to open.

Valid Usage (Implicit)

- VUID-vkQueueBeginDebugUtilsLabelEXT-queue-parameter queue must be a valid VkQueue handle
- VUID-vkQueueBeginDebugUtilsLabelEXT-pLabelInfo-parameter pLabelInfo must be a valid pointer to a valid VkDebugUtilsLabelEXT structure

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>

The VkDebugUtilsLabelEXT structure is defined as:

// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsLabelEXT {
    VkStructureType sType;
    const void* pNext;
    const char* pLabelName;
    float color[4];
} VkDebugUtilsLabelEXT;

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- pLabelName is a pointer to a null-terminated UTF-8 string containing the name of the label.
- color is an optional RGBA color value that can be associated with the label. A particular implementation may choose to ignore this color value. The values contain RGBA values in order, in the range 0.0 to 1.0. If all elements in color are set to 0.0 then it is ignored.
Valid Usage (Implicit)

- **VUID-VkDebugUtilsLabelEXT-sType-sType**
  
  `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT`

- **VUID-VkDebugUtilsLabelEXT-pNext-pNext**
  
  `pNext` **must** be `NULL`

- **VUID-VkDebugUtilsLabelEXT-pLabelName-parameter**
  
  `pLabelName` **must** be a null-terminated UTF-8 string

A queue debug label region is closed by calling:

```c
// Provided by VK_EXT_debug_utils
void vkQueueEndDebugUtilsLabelEXT(
    VkQueue queue);
```

- `queue` is the queue in which a debug label region should be closed.

The calls to `vkQueueBeginDebugUtilsLabelEXT` and `vkQueueEndDebugUtilsLabelEXT` **must** be matched and balanced.

Valid Usage

- **VUID-vkQueueEndDebugUtilsLabelEXT-None-01911**
  
  There **must** be an outstanding `vkQueueBeginDebugUtilsLabelEXT` command prior to the `vkQueueEndDebugUtilsLabelEXT` on the queue

Valid Usage (Implicit)

- **VUID-vkQueueEndDebugUtilsLabelEXT-queue-parameter**
  
  `queue` **must** be a valid `VkQueue` handle

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>

A single label can be inserted into a queue by calling:
// Provided by VK_EXT_debug_utils
void vkQueueInsertDebugUtilsLabelEXT(
    VkQueue queue,
    const VkDebugUtilsLabelEXT* pLabelInfo);

- `queue` is the queue into which a debug label will be inserted.
- `pLabelInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label to insert.

### Valid Usage (Implicit)

- VUID-vkQueueInsertDebugUtilsLabelEXT-queue-parameter
  - `queue` must be a valid `VkQueue` handle
- VUID-vkQueueInsertDebugUtilsLabelEXT-pLabelInfo-parameter
  - `pLabelInfo` must be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>Any</td>
<td>-</td>
</tr>
</tbody>
</table>

#### 45.1.3. Command Buffer Labels

Typical Vulkan applications will submit many command buffers in each frame, with each command buffer containing a large number of individual commands. Being able to logically annotate regions of command buffers that belong together as well as hierarchically subdivide the frame is important to a developer's ability to navigate the commands viewed holistically.

To identify the beginning of a debug label region in a command buffer, `vkCmdBeginDebugUtilsLabelEXT` can be used as defined below.

To indicate the end of a debug label region in a command buffer, `vkCmdEndDebugUtilsLabelEXT` can be used.

To insert a single command buffer debug label inside of a command buffer, `vkCmdInsertDebugUtilsLabelEXT` can be used as defined below.

A command buffer debug label region can be opened by calling:
// Provided by VK_EXT_debug_utils
void vkCmdBeginDebugUtilsLabelEXT(
    VkCommandBuffer                             commandBuffer,
    const VkDebugUtilsLabelEXT*                 pLabelInfo);

- `commandBuffer` is the command buffer into which the command is recorded.
- `pLabelInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label region to open.

Valid Usage (Implicit)

- `VUID-vkCmdBeginDebugUtilsLabelEXT-commandBuffer-parameter` | `commandBuffer` must be a valid `VkCommandBuffer` handle
- `VUID-vkCmdBeginDebugUtilsLabelEXT-pLabelInfo-parameter` | `pLabelInfo` must be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure
- `VUID-vkCmdBeginDebugUtilsLabelEXT-commandBuffer-recording` | `commandBuffer` must be in the recording state
- `VUID-vkCmdBeginDebugUtilsLabelEXT-commandBuffer-cmdpool` | The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td>Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A command buffer label region can be closed by calling:

// Provided by VK_EXT_debug_utils
void vkCmdEndDebugUtilsLabelEXT(
    VkCommandBuffer                             commandBuffer);
• `commandBuffer` is the command buffer into which the command is recorded.

An application may open a debug label region in one command buffer and close it in another, or otherwise split debug label regions across multiple command buffers or multiple queue submissions. When viewed from the linear series of submissions to a single queue, the calls to `vkCmdBeginDebugUtilsLabelEXT` and `vkCmdEndDebugUtilsLabelEXT` must be matched and balanced.

### Valid Usage

- **VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-01912**
  There must be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command prior to the `vkCmdEndDebugUtilsLabelEXT` on the queue that `commandBuffer` is submitted to.

- **VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-01913**
  If `commandBuffer` is a secondary command buffer, there must be an outstanding `vkCmdBeginDebugUtilsLabelEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdEndDebugUtilsLabelEXT`.

### Valid Usage (Implicit)

- **VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-parameter**
  `commandBuffer` must be a valid `VkCommandBuffer` handle.

- **VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-recording**
  `commandBuffer` must be in the recording state.

- **VUID-vkCmdEndDebugUtilsLabelEXT-commandBuffer-cmdpool**
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations.

### Host Synchronization

- Host access to `commandBuffer` must be externally synchronized.

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized.

### Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>
A single debug label can be inserted into a command buffer by calling:

```c
// Provided by VK_EXT_debug_utils
void vkCmdInsertDebugUtilsLabelEXT(
    VkCommandBuffer                             commandBuffer,
    const VkDebugUtilsLabelEXT*                 pLabelInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pInfo` is a pointer to a `VkDebugUtilsLabelEXT` structure specifying parameters of the label to insert.

**Valid Usage (Implicit)**

- VUID-vkCmdInsertDebugUtilsLabelEXT-commandBuffer-parameter
  `commandBuffer` must be a valid `VkCommandBuffer` handle
- VUID-vkCmdInsertDebugUtilsLabelEXT-pLabelInfo-parameter
  `pLabelInfo` must be a valid pointer to a valid `VkDebugUtilsLabelEXT` structure
- VUID-vkCmdInsertDebugUtilsLabelEXT-commandBuffer-recording
  `commandBuffer` must be in the recording state
- VUID-vkCmdInsertDebugUtilsLabelEXT-commandBuffer-cmdpool
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

**Host Synchronization**

- Host access to `commandBuffer` must be externally synchronized
- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

**Command Properties**

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td>Both</td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

### 45.1.4. Debug Messengers

Vulkan allows an application to register multiple callbacks with any Vulkan component wishing to report debug information. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. A primary producer of callback messages
are the validation layers. An application can register callbacks even when no validation layers are enabled, but they will only be called for the Vulkan loader and, if implemented, other layer and driver events.

A VkDebugUtilsMessengerEXT is a messenger object which handles passing along debug messages to a provided debug callback.

```
// Provided by VK_EXT_debug_utils
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDebugUtilsMessengerEXT)
```

The debug messenger will provide detailed feedback on the application's use of Vulkan when events of interest occur. When an event of interest does occur, the debug messenger will submit a debug message to the debug callback that was provided during its creation. Additionally, the debug messenger is responsible with filtering out debug messages that the callback is not interested in and will only provide desired debug messages.

A debug messenger triggers a debug callback with a debug message when an event of interest occurs. To create a debug messenger which will trigger a debug callback, call:

```
// Provided by VK_EXT_debug_utils
VkResult vkCreateDebugUtilsMessengerEXT(
    VkInstance                                  instance,
    const VkDebugUtilsMessengerCreateInfoEXT*   pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDebugUtilsMessengerEXT*                   pMessenger);
```

- **instance** is the instance the messenger will be used with.
- **pCreateInfo** is a pointer to a VkDebugUtilsMessengerCreateInfoEXT structure containing the callback pointer, as well as defining conditions under which this messenger will trigger the callback.
- **pAllocator** controls host memory allocation as described in the Memory Allocation chapter.
- **pMessenger** is a pointer to a VkDebugUtilsMessengerEXT handle in which the created object is returned.
Valid Usage (Implicit)

- **VUID-vkCreateDebugUtilsMessengerEXT-instance-parameter**
  
  *instance* must be a valid *VkInstance* handle

- **VUID-vkCreateDebugUtilsMessengerEXT-pCreateInfo-parameter**
  
  *pCreateInfo* must be a valid pointer to a valid *VkDebugUtilsMessengerCreateInfoEXT* structure

- **VUID-vkCreateDebugUtilsMessengerEXT-pAllocator-parameter**
  
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkCreateDebugUtilsMessengerEXT-pMessenger-parameter**
  
  *pMessenger* must be a valid pointer to a *VkDebugUtilsMessengerEXT* handle

Return Codes

**Success**

- **VK_SUCCESS**

**Failure**

- **VK_ERROR_OUT_OF_HOST_MEMORY**

The application must ensure that *vkCreateDebugUtilsMessengerEXT* is not executed in parallel with any Vulkan command that is also called with *instance* or child of *instance* as the dispatchable argument.

The definition of *VkDebugUtilsMessengerCreateInfoEXT* is:

```c
// Provided by VK_EXT_debug_utils
typedef struct VkDebugUtilsMessengerCreateInfoEXT {
    VkStructureType sType;        
    const void*    pNext;       
    VkDebugUtilsMessengerCreateFlagsEXT flags;       
    VkDebugUtilsMessageSeverityFlagsEXT messageSeverity; 
    VkDebugUtilsMessageTypeFlagsEXT messageType;     
    PFN_vkDebugUtilsMessengerCallbackEXT pfnUserCallback; 
    void*               pUserData;   
} VkDebugUtilsMessengerCreateInfoEXT; 
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is 0 and is reserved for future use.
- **messageSeverity** is a bitmask of *VkDebugUtilsMessageSeverityFlagBitsEXT* specifying which severity of event(s) will cause this callback to be called.
• **messageType** is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) will cause this callback to be called.

• **pfnUserCallback** is the application callback function to call.

• **pUserData** is user data to be passed to the callback.

For each `VkDebugUtilsMessengerEXT` that is created the `VkDebugUtilsMessengerCreateInfoEXT::messageSeverity` and `VkDebugUtilsMessengerCreateInfoEXT::messageType` determine when that `VkDebugUtilsMessengerCreateInfoEXT::pfnUserCallback` is called. The process to determine if the user's `pfnUserCallback` is triggered when an event occurs is as follows:

1. The implementation will perform a bitwise AND of the event's `VkDebugUtilsMessageSeverityFlagBitsEXT` with the `messageSeverity` provided during creation of the `VkDebugUtilsMessengerEXT` object.
   a. If the value is 0, the message is skipped.

2. The implementation will perform bitwise AND of the event's `VkDebugUtilsMessageTypeFlagBitsEXT` with the `messageType` provided during the creation of the `VkDebugUtilsMessengerEXT` object.
   a. If the value is 0, the message is skipped.

3. The callback will trigger a debug message for the current event

The callback will come directly from the component that detected the event, unless some other layer intercepts the calls for its own purposes (filter them in a different way, log to a system error log, etc.).

An application can receive multiple callbacks if multiple `VkDebugUtilsMessengerEXT` objects are created. A callback will always be executed in the same thread as the originating Vulkan call.

A callback can be called from multiple threads simultaneously (if the application is making Vulkan calls from multiple threads).

<table>
<thead>
<tr>
<th>Valid Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>• VUID-VkDebugUtilsMessengerCreateInfoEXT-pfnUserCallback-01914</td>
</tr>
<tr>
<td><strong>pfnUserCallback</strong> must be a valid <code>PFN_vkDebugUtilsMessengerCallbackEXT</code></td>
</tr>
</tbody>
</table>
Valid Usage (Implicit)

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-sType-sType**
  
  *sType must be VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENER_CREATE_INFO_EXT*

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-flags-zero bitmask**
  
  *flags must be 0*

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-messageSeverity-parameter**
  
  *messageSeverity must be a valid combination of VkDebugUtilsMessageSeverityFlagBitsEXT values*

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-messageSeverity-requiredbitmask**
  
  *messageSeverity must not be 0*

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-messageType-parameter**
  
  *messageType must be a valid combination of VkDebugUtilsMessageTypeFlagBitsEXT values*

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-messageType-requiredbitmask**
  
  *messageType must not be 0*

- **VUID-VkDebugUtilsMessengerCreateInfoEXT-pfnUserCallback-parameter**
  
  *pfUserCallback must be a valid PFN_vkDebugUtilsMessengerCallbackEXT value*

---

// Provided by VK_EXT_debug_utils

typedef VkFlags VkDebugUtilsMessengerCreateFlagsEXT;

VkDebugUtilsMessengerCreateFlagsEXT is a bitmask type for setting a mask, but is currently reserved for future use.

Bits which can be set in VkDebugUtilsMessengerCreateInfoEXT::messageSeverity, specifying event severities which cause a debug messenger to call the callback, are:

// Provided by VK_EXT_debug_utils

typedef enum VkDebugUtilsMessageSeverityFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT = 0x00000010,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT = 0x00000100,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT = 0x00001000,
} VkDebugUtilsMessageSeverityFlagBitsEXT;

- **VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT** specifies the most verbose output indicating all diagnostic messages from the Vulkan loader, layers, and drivers should be captured.

- **VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT** specifies an informational message such as resource details that may be handy when debugging an application.

- **VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT** specifies use of Vulkan that may expose an app bug. Such cases may not be immediately harmful, such as a fragment shader outputting to a location with no attachment. Other cases may point to behavior that is almost certainly bad
when unintended such as using an image whose memory has not been filled. In general if you see a warning but you know that the behavior is intended/desired, then simply ignore the warning.

- **VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT** specifies that the application has violated a valid usage condition of the specification.

**Note**

The values of `VkDebugUtilsMessageSeverityFlagBitsEXT` are sorted based on severity. The higher the flag value, the more severe the message. This allows for simple boolean operation comparisons when looking at `VkDebugUtilsMessageSeverityFlagBitsEXT` values.

For example:

```c
if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    // Do something for warnings and errors
}
```

In addition, space has been left between the enums to allow for later addition of new severities in between the existing values.

```c
// Provided by VK_EXT_debug_utils
typedef VkFlags VkDebugUtilsMessageSeverityFlagsEXT;
```

`VkDebugUtilsMessageSeverityFlagsEXT` is a bitmask type for setting a mask of zero or more `VkDebugUtilsMessageSeverityFlagBitsEXT`.

Bits which **can** be set in `VkDebugUtilsMessengerCreateInfoEXT::messageType`, specifying event types which cause a debug messenger to call the callback, are:

```c
// Provided by VK_EXT_debug_utils
typedef enum VkDebugUtilsMessageTypeFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT = 0x00000002,
    VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT = 0x00000004,
} VkDebugUtilsMessageTypeFlagBitsEXT;
```

- **VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT** specifies that some general event has occurred. This is typically a non-specification, non-performance event.
- **VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT** specifies that something has occurred during validation against the Vulkan specification that may indicate invalid behavior.
- **VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT** specifies a potentially non-optimal use of Vulkan, e.g. using `vkCmdClearColorImage` when setting `VkAttachmentDescription::loadOp` to `VK_ATTACHMENT_LOAD_OP_CLEAR` would have worked.
\textbf{VkDebugUtilsMessageTypeFlagsEXT} is a bitmask type for setting a mask of zero or more \textbf{VkDebugUtilsMessageTypeFlagBitsEXT}.

The prototype for the \textbf{VkDebugUtilsMessengerCreateInfoEXT::pfnUserCallback} function implemented by the application is:

\begin{verbatim}
// Provided by VK_EXT_debug_utils
typedef VkFlags VkDebugUtilsMessageTypeFlagsEXT;

typedef VkBool32 (VKAPI_PTR *PFN_vkDebugUtilsMessengerCallbackEXT)(
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageTypes,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
    void* pUserData);
\end{verbatim}

- \texttt{messageSeverity} specifies the \textbf{VkDebugUtilsMessageSeverityFlagBitsEXT} that triggered this callback.
- \texttt{messageTypes} is a bitmask of \textbf{VkDebugUtilsMessageTypeFlagBitsEXT} specifying which type of event(s) triggered this callback.
- \texttt{pCallbackData} contains all the callback related data in the \textbf{VkDebugUtilsMessengerCallbackDataEXT} structure.
- \texttt{pUserData} is the user data provided when the \textbf{VkDebugUtilsMessengerEXT} was created.

The callback returns a \textbf{VkBool32}, which is interpreted in a layer-specified manner. The application should always return \texttt{VK_FALSE}. The \texttt{VK_TRUE} value is reserved for use in layer development.

\textbf{Valid Usage}

- VUID-PFN_vkDebugUtilsMessengerCallbackEXT-None-04769
  The callback must not make calls to any Vulkan commands.

The definition of \textbf{VkDebugUtilsMessengerCallbackDataEXT} is:
typedef struct VkDebugUtilsMessengerCallbackDataEXT {
    VkStructureType sType;
    const void* pNext;
    VkDebugUtilsMessengerCallbackDataFlagsEXT flags;
    const char* pMessageIdName;
    int32_t messageIdNumber;
    const char* pMessage;
    uint32_t queueLabelCount;
    const VkDebugUtilsLabelEXT* pQueueLabels;
    uint32_t cmdBufLabelCount;
    const VkDebugUtilsLabelEXT* pCmdBufLabels;
    uint32_t objectCount;
    const VkDebugUtilsObjectNameInfoEXT* pObjects;
} VkDebugUtilsMessengerCallbackDataEXT;

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is 0 and is reserved for future use.
- **pMessageIdName** is a null-terminated string that identifies the particular message ID that is associated with the provided message. If the message corresponds to a validation layer message, then this string may contain the portion of the Vulkan specification that is believed to have been violated.
- **messageIdNumber** is the ID number of the triggering message. If the message corresponds to a validation layer message, then this number is related to the internal number associated with the message being triggered.
- **pMessage** is a null-terminated string detailing the trigger conditions.
- **queueLabelCount** is a count of items contained in the **pQueueLabels** array.
- **pQueueLabels** is NULL or a pointer to an array of VkDebugUtilsLabelEXT active in the current VkQueue at the time the callback was triggered. Refer to Queue Labels for more information.
- **cmdBufLabelCount** is a count of items contained in the **pCmdBufLabels** array.
- **pCmdBufLabels** is NULL or a pointer to an array of VkDebugUtilsLabelEXT active in the current VkCommandBuffer at the time the callback was triggered. Refer to Command Buffer Labels for more information.
- **objectCount** is a count of items contained in the **pObjects** array.
- **pObjects** is a pointer to an array of VkDebugUtilsObjectNameInfoEXT objects related to the detected issue. The array is roughly in order or importance, but the 0th element is always guaranteed to be the most important object for this message.

**Note**
This structure should only be considered valid during the lifetime of the triggered callback.
Since adding queue and command buffer labels behaves like pushing and popping onto a stack, the order of both `pQueueLabels` and `pCmdBufLabels` is based on the order the labels were defined. The result is that the first label in either `pQueueLabels` or `pCmdBufLabels` will be the first defined (and therefore the oldest) while the last label in each list will be the most recent.

**Note**

`pQueueLabels` will only be non-NULL if one of the objects in `pObjects` can be related directly to a defined `VkQueue` which has had one or more labels associated with it.

Likewise, `pCmdBufLabels` will only be non-NULL if one of the objects in `pObjects` can be related directly to a defined `VkCommandBuffer` which has had one or more labels associated with it. Additionally, while command buffer labels allow for beginning and ending across different command buffers, the debug messaging framework cannot guarantee that labels in `pCmdBufLabels` will contain those defined outside of the associated command buffer. This is partially due to the fact that the association of one command buffer with another may not have been defined at the time the debug message is triggered.

**Valid Usage (Implicit)**

- `VUID-VkDebugUtilsMessengerCallbackDataEXT-sType-sType`
  - `sType` must be `VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT`
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-pNext-pNext`
  - `pNext` must be `NULL`
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-flags-zerobitmask`
  - `flags` must be `0`
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-pMessageIdName-parameter`
  - If `pMessageIdName` is not `NULL`, `pMessageIdName` must be a null-terminated UTF-8 string
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-pMessage-parameter`
  - `pMessage` must be a null-terminated UTF-8 string
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-pQueueLabels-parameter`
  - If `queueLabelCount` is not `0`, `pQueueLabels` must be a valid pointer to an array of `queueLabelCount` valid `VkDebugUtilsLabelEXT` structures
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-pCmdBufLabels-parameter`
  - If `cmdBufLabelCount` is not `0`, `pCmdBufLabels` must be a valid pointer to an array of `cmdBufLabelCount` valid `VkDebugUtilsLabelEXT` structures
- `VUID-VkDebugUtilsMessengerCallbackDataEXT-pObjects-parameter`
  - If `objectCount` is not `0`, `pObjects` must be a valid pointer to an array of `objectCount` valid `VkDebugUtilsObjectNameInfoEXT` structures

```c
// Provided by VK_EXT_debug_utils
typedef VkFlags VkDebugUtilsMessengerCallbackDataFlagsEXT;
```
VkDebugUtilsMessengerCallbackDataFlagsEXT is a bitmask type for setting a mask, but is currently reserved for future use.

There may be times that a user wishes to intentionally submit a debug message. To do this, call:

```c
// Provided by VK_EXT_debug_utils
void vkSubmitDebugUtilsMessageEXT(
    VkInstance                                  instance,
    VkDebugUtilsMessageSeverityFlagBitsEXT      messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT             messageTypes,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData);
```

- `instance` is the debug stream's `VkInstance`.
- `messageSeverity` is the `VkDebugUtilsMessageSeverityFlagBitsEXT` severity of this event/message.
- `messageTypes` is a bitmask of `VkDebugUtilsMessageTypeFlagBitsEXT` specifying which type of event(s) to identify with this message.
- `pCallbackData` contains all the callback related data in the `VkDebugUtilsMessengerCallbackDataEXT` structure.

The call will propagate through the layers and generate callback(s) as indicated by the message’s flags. The parameters are passed on to the callback in addition to the `pUserData` value that was defined at the time the messenger was registered.

**Valid Usage**

- VUID-vkSubmitDebugUtilsMessageEXT-objectType-02591
  The `objectType` member of each element of `pCallbackData->pObjects` must not be `VK_OBJECT_TYPE_UNKNOWN`

**Valid Usage (Implicit)**

- VUID-vkSubmitDebugUtilsMessageEXT-instance-parameter
  `instance` must be a valid `VkInstance` handle
- VUID-vkSubmitDebugUtilsMessageEXT-messageSeverity-parameter
  `messageSeverity` must be a valid `VkDebugUtilsMessageSeverityFlagBitsEXT` value
- VUID-vkSubmitDebugUtilsMessageEXT-messageTypes-parameter
  `messageTypes` must be a valid combination of `VkDebugUtilsMessageTypeFlagBitsEXT` values
- VUID-vkSubmitDebugUtilsMessageEXT-messageTypes-requiredbitmap
  `messageTypes` must not be 0
- VUID-vkSubmitDebugUtilsMessageEXT-pCallbackData-parameter
  `pCallbackData` must be a valid pointer to a valid `VkDebugUtilsMessengerCallbackDataEXT` structure
To destroy a `VkDebugUtilsMessengerEXT` object, call:

```c
// Provided by VK_EXT_debug_utils
void vkDestroyDebugUtilsMessengerEXT(
    VkInstance                                  instance,   
    VkDebugUtilsMessengerEXT                    messenger,   
    const VkAllocationCallbacks*                pAllocator);
```

- `instance` is the instance where the callback was created.
- `messenger` is the `VkDebugUtilsMessengerEXT` object to destroy. `messenger` is an externally synchronized object and must not be used on more than one thread at a time. This means that `vkDestroyDebugUtilsMessengerEXT` must not be called when a callback is active.
- `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

### Valid Usage

- **VUID-vkDestroyDebugUtilsMessengerEXT-messenger-01915**
  If `VkAllocationCallbacks` were provided when `messenger` was created, a compatible set of callbacks must be provided here.

- **VUID-vkDestroyDebugUtilsMessengerEXT-messenger-01916**
  If no `VkAllocationCallbacks` were provided when `messenger` was created, `pAllocator` must be `NULL`.

### Valid Usage (Implicit)

- **VUID-vkDestroyDebugUtilsMessengerEXT-instance-parameter**
  `instance` must be a valid `VkInstance` handle.

- **VUID-vkDestroyDebugUtilsMessengerEXT-messenger-parameter**
  If `messenger` is not `VK_NULL_HANDLE`, `messenger` must be a valid `VkDebugUtilsMessengerEXT` handle.

- **VUID-vkDestroyDebugUtilsMessengerEXT-pAllocator-parameter**
  If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

- **VUID-vkDestroyDebugUtilsMessengerEXT-messenger-parent**
  If `messenger` is a valid handle, it must have been created, allocated, or retrieved from `instance`.

### Host Synchronization

- Host access to `messenger` must be externally synchronized.

The application must ensure that `vkDestroyDebugUtilsMessengerEXT` is not executed in parallel.
with any Vulkan command that is also called with instance or child of instance as the dispatchable argument.

### 45.2. Debug Markers

Debug markers provide a flexible way for debugging and validation layers to receive annotation and debug information.

The **Object Annotation** section describes how to associate a name or binary data with a Vulkan object.

The **Command Buffer Markers** section describes how to associate logical elements of the scene with commands in the command buffer.

#### 45.2.1. Object Annotation

The commands in this section allow application developers to associate user-defined information with Vulkan objects at will.

An object can be given a user-friendly name by calling:

```c
// Provided by VK_EXT_debug_marker
VkResult vkDebugMarkerSetObjectNameEXT(
    VkDevice                                    device,
    const VkDebugMarkerObjectNameInfoEXT*       pNameInfo);
```

- **device** is the device that created the object.
- **pNameInfo** is a pointer to a `VkDebugMarkerObjectNameInfoEXT` structure specifying the parameters of the name to set on the object.

**Valid Usage (Implicit)**

- VUID-vkDebugMarkerSetObjectNameEXT-device-parameter
  - `device` must be a valid `VkDevice` handle
- VUID-vkDebugMarkerSetObjectNameEXT-pNameInfo-parameter
  - `pNameInfo` must be a valid pointer to a valid `VkDebugMarkerObjectNameInfoEXT` structure

**Host Synchronization**

- Host access to `pNameInfo->object` must be externally synchronized
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The VkDebugMarkerObjectNameInfoEXT structure is defined as:

```c
// Provided by VK_EXT_debug_marker
typedef struct VkDebugMarkerObjectNameInfoEXT {
    VkStructureType               sType;
    const void*                   pNext;
    VkDebugReportObjectTypeEXT    objectType;
    uint64_t                      object;
    const char*                   pObjectName;
} VkDebugMarkerObjectNameInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `objectType` is a `VkDebugReportObjectTypeEXT` specifying the type of the object to be named.
- `object` is the object to be named.
- `pObjectName` is a null-terminated UTF-8 string specifying the name to apply to `object`.

Applications **may** change the name associated with an object simply by calling `vkDebugMarkerSetObjectNameEXT` again with a new string. To remove a previously set name, `pObjectName` **should** be set to an empty string.

Valid Usage

- VUID-VkDebugMarkerObjectNameInfoEXT-objectType-01490
  `objectType` **must not be** `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT`

- VUID-VkDebugMarkerObjectNameInfoEXT-object-01491
  `object` **must not be** `VK_NULL_HANDLE`

- VUID-VkDebugMarkerObjectNameInfoEXT-object-01492
  `object` **must be** a Vulkan object of the type associated with `objectType` as defined in `VkDebugReportObjectTypeEXT` and `Vulkan Handle Relationship`
Valid Usage (Implicit)

- **VUID-VkDebugMarkerObjectNameInfoEXT-sType-sType**
  
  \texttt{sType} \textbf{must be} \texttt{VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT}

- **VUID-VkDebugMarkerObjectNameInfoEXT-pNext-pNext**
  
  \texttt{pNext} \textbf{must be} \texttt{NULL}

- **VUID-VkDebugMarkerObjectNameInfoEXT-objectType-parameter**
  
  \texttt{objectType} \textbf{must be} a valid \texttt{VkDebugReportObjectTypeEXT} value

- **VUID-VkDebugMarkerObjectNameInfoEXT-pObjectName-parameter**
  
  \texttt{pObjectName} \textbf{must be} a null-terminated UTF-8 string

In addition to setting a name for an object, debugging and validation layers may have uses for additional binary data on a per-object basis that has no other place in the Vulkan API. For example, a \texttt{VkShaderModule} could have additional debugging data attached to it to aid in offline shader tracing. To attach data to an object, call:

```c
// Provided by VK_EXT_debug_marker
VkResult vkDebugMarkerSetObjectTagEXT(
    VkDevice                                    device,
    const VkDebugMarkerObjectTagInfoEXT*        pTagInfo);
```

- \texttt{device} \textbf{is} the device that created the object.

- \texttt{pTagInfo} \textbf{is} a pointer to a \texttt{VkDebugMarkerObjectTagInfoEXT} structure specifying the parameters of the tag to attach to the object.

Valid Usage (Implicit)

- **VUID-vkDebugMarkerSetObjectTagEXT-device-parameter**
  
  \texttt{device} \textbf{must be} a valid \texttt{VkDevice} handle

- **VUID-vkDebugMarkerSetObjectTagEXT-pTagInfo-parameter**
  
  \texttt{pTagInfo} \textbf{must be} a valid pointer to a valid \texttt{VkDebugMarkerObjectTagInfoEXT} structure

Host Synchronization

- Host access to \texttt{pTagInfo->object} \textbf{must be} externally synchronized
Return Codes

Success
- VK_SUCCESS

Failure
- VK_ERROR_OUT_OF_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

The `VkDebugMarkerObjectTagInfoEXT` structure is defined as:

```c
// Provided by VK_EXT_debug_marker
typedef struct VkDebugMarkerObjectTagInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDebugReportObjectTypeEXT objectType;
    uint64_t object;
    uint64_t tagName;
    size_t tagSize;
    const void* pTag;
} VkDebugMarkerObjectTagInfoEXT;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to a structure extending this structure.
- `objectType` is a `VkDebugReportObjectTypeEXT` specifying the type of the object to be named.
- `object` is the object to be tagged.
- `tagName` is a numerical identifier of the tag.
- `tagSize` is the number of bytes of data to attach to the object.
- `pTag` is a pointer to an array of `tagSize` bytes containing the data to be associated with the object.

The `tagName` parameter gives a name or identifier to the type of data being tagged. This can be used by debugging layers to easily filter for only data that can be used by that implementation.

Valid Usage

- VUID-VkDebugMarkerObjectTagInfoEXT-objectType-01493
  `objectType` must not be `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT`
- VUID-VkDebugMarkerObjectTagInfoEXT-object-01494
  `object` must not be `VK_NULL_HANDLE`
- VUID-VkDebugMarkerObjectTagInfoEXT-object-01495
  `object` must be a Vulkan object of the type associated with `objectType` as defined in `VkDebugReportObjectTypeEXT` and `Vulkan Handle Relationship`
Valid Usage (Implicit)

- VUID-VkDebugMarkerObjectTagInfoEXT-sType-sType
  
  `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_TAG_INFO_EXT`

- VUID-VkDebugMarkerObjectTagInfoEXT-pNext-pNext
  
  `pNext` **must** be `NULL`

- VUID-VkDebugMarkerObjectTagInfoEXT-objectType-parameter
  
  `objectType` **must** be a valid `VkDebugReportObjectTypeEXT` value

- VUID-VkDebugMarkerObjectTagInfoEXT-pTag-parameter
  
  `pTag` **must** be a valid pointer to an array of `tagSize` bytes

- VUID-VkDebugMarkerObjectTagInfoEXT-tagSize-arraylength
  
  `tagSize` **must** be greater than `0`

45.2.2. Command Buffer Markers

Typical Vulkan applications will submit many command buffers in each frame, with each command buffer containing a large number of individual commands. Being able to logically annotate regions of command buffers that belong together as well as hierarchically subdivide the frame is important to a developer’s ability to navigate the commands viewed holistically.

The marker commands `vkCmdDebugMarkerBeginEXT` and `vkCmdDebugMarkerEndEXT` define regions of a series of commands that are grouped together, and they can be nested to create a hierarchy. The `vkCmdDebugMarkerInsertEXT` command allows insertion of a single label within a command buffer.

A marker region can be opened by calling:

```c
// Provided by VK_EXT_debug_marker
define vkCmdDebugMarkerBeginEXT(
    VkCommandBuffer                             commandBuffer,  
    const VkDebugMarkerMarkerInfoEXT*           pMarkerInfo);
```

- `commandBuffer` is the command buffer into which the command is recorded.
- `pMarkerInfo` is a pointer to a `VkDebugMarkerMarkerInfoEXT` structure specifying the parameters of the marker region to open.
Valid Usage (Implicit)

- VUID-vkCmdDebugMarkerBeginEXT-commandBuffer-parameter
  commandBuffer must be a valid VkCommandBuffer handle

- VUID-vkCmdDebugMarkerBeginEXT-pMarkerInfo-parameter
  pMarkerInfo must be a valid pointer to a valid VkDebugMarkerMarkerInfoEXT structure

- VUID-vkCmdDebugMarkerBeginEXT-commandBuffer-recording
  commandBuffer must be in the recording state

- VUID-vkCmdDebugMarkerBeginEXT-commandBuffer-cmdpool
  The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations

Host Synchronization

- Host access to commandBuffer must be externally synchronized
- Host access to the VkCommandPool that commandBuffer was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

The VkDebugMarkerMarkerInfoEXT structure is defined as:

```c
// Provided by VK_EXT_debug_marker
typedef struct VkDebugMarkerMarkerInfoEXT {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pMarkerName;
    float              color[4];
} VkDebugMarkerMarkerInfoEXT;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- pMarkerName is a pointer to a null-terminated UTF-8 string containing the name of the marker.
- color is an optional RGBA color value that can be associated with the marker. A particular implementation may choose to ignore this color value. The values contain RGBA values in
order, in the range 0.0 to 1.0. If all elements in `color` are set to 0.0 then it is ignored.

### Valid Usage (Implicit)

- **VUID-VkDebugMarkerMarkerInfoEXT-sType-sType**  
  `sType` **must** be `VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT`

- **VUID-VkDebugMarkerMarkerInfoEXT-pNext-pNext**  
  `pNext` **must** be `NULL`

- **VUID-VkDebugMarkerMarkerInfoEXT-pMarkerName-parameter**  
  `pMarkerName` **must** be a null-terminated UTF-8 string

A marker region can be closed by calling:

```c
// Provided by VK_EXT_debug_marker
void vkCmdDebugMarkerEndEXT(
    VkCommandBuffer commandBuffer);
```

- `commandBuffer` is the command buffer into which the command is recorded.

An application **may** open a marker region in one command buffer and close it in another, or otherwise split marker regions across multiple command buffers or multiple queue submissions. When viewed from the linear series of submissions to a single queue, the calls to `vkCmdDebugMarkerBeginEXT` and `vkCmdDebugMarkerEndEXT` **must** be matched and balanced.

### Valid Usage

- **VUID-vkCmdDebugMarkerEndEXT-commandBuffer-01239**  
  There **must** be an outstanding `vkCmdDebugMarkerBeginEXT` command prior to the `vkCmdDebugMarkerEndEXT` on the queue that `commandBuffer` is submitted to

- **VUID-vkCmdDebugMarkerEndEXT-commandBuffer-01240**  
  If `commandBuffer` is a secondary command buffer, there **must** be an outstanding `vkCmdDebugMarkerBeginEXT` command recorded to `commandBuffer` that has not previously been ended by a call to `vkCmdDebugMarkerEndEXT`
Valid Usage (Implicit)

- VUID-vkCmdDebugMarkerEndEXT-commandBuffer-parameter
  \(\text{commandBuffer must be a valid } \text{VkCommandBuffer handle}\)
- VUID-vkCmdDebugMarkerEndEXT-commandBuffer-recording
  \(\text{commandBuffer must be in the recording state}\)
- VUID-vkCmdDebugMarkerEndEXT-commandBuffer-cmdpool
  The \text{VkCommandPool} that \text{commandBuffer} was allocated from \textbf{must} support graphics, or compute operations

Host Synchronization

- Host access to \text{commandBuffer} \textbf{must} be externally synchronized
- Host access to the \text{VkCommandPool} that \text{commandBuffer} was allocated from \textbf{must} be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

A single marker label can be inserted into a command buffer by calling:

```c
// Provided by VK_EXT_debug_marker
void vkCmdDebugMarkerInsertEXT(
  VkCommandBuffer              commandBuffer,
  const VkDebugMarkerMarkerInfoEXT* pMarkerInfo);
```

- \text{commandBuffer} is the command buffer into which the command is recorded.
- \text{pMarkerInfo} is a pointer to a \text{VkDebugMarkerMarkerInfoEXT} structure specifying the parameters of the marker to insert.
Valid Usage (Implicit)

- VUID-vkCmdDebugMarkerInsertEXT-commandBuffer-parameter
  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdDebugMarkerInsertEXT-pMarkerInfo-parameter
  
  `pMarkerInfo` must be a valid pointer to a valid `VkDebugMarkerMarkerInfoEXT` structure

- VUID-vkCmdDebugMarkerInsertEXT-commandBuffer-recording
  
  `commandBuffer` must be in the recording state

- VUID-vkCmdDebugMarkerInsertEXT-commandBuffer-cmdpool
  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

45.3. Debug Report Callbacks

Debug report callbacks are represented by `VkDebugReportCallbackEXT` handles:

```c
// Provided by VK_EXT_debug_report
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDebugReportCallbackEXT)
```

Debug report callbacks give more detailed feedback on the application's use of Vulkan when events of interest occur.

To register a debug report callback, an application uses `vkCreateDebugReportCallbackEXT`. 
// Provided by VK_EXT_debug_report
VkResult vkCreateDebugReportCallbackEXT(
    VkInstance instance,
    const VkDebugReportCallbackCreateInfoEXT* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDebugReportCallbackEXT* pCallback);

• \texttt{instance} is the instance the callback will be logged on.
• \texttt{pCreateInfo} is a pointer to a \texttt{VkDebugReportCallbackCreateInfoEXT} structure defining the conditions under which this callback will be called.
• \texttt{pAllocator} controls host memory allocation as described in the Memory Allocation chapter.
• \texttt{pCallback} is a pointer to a \texttt{VkDebugReportCallbackEXT} handle in which the created object is returned.

Valid Usage (Implicit)

• VUID-vkCreateDebugReportCallbackEXT-instance-parameter \texttt{instance} must be a valid \texttt{VkInstance} handle
• VUID-vkCreateDebugReportCallbackEXT-pCreateInfo-parameter \texttt{pCreateInfo} must be a valid pointer to a valid \texttt{VkDebugReportCallbackCreateInfoEXT} structure
• VUID-vkCreateDebugReportCallbackEXT-pAllocator-parameter If \texttt{pAllocator} is not \texttt{NULL}, \texttt{pAllocator} must be a valid pointer to a valid \texttt{VkAllocationCallbacks} structure
• VUID-vkCreateDebugReportCallbackEXT-pCallback-parameter \texttt{pCallback} must be a valid pointer to a \texttt{VkDebugReportCallbackEXT} handle

Return Codes

Success
• VK_SUCCESS

Failure
• VK_ERROR_OUT_OF_HOST_MEMORY

The definition of \texttt{VkDebugReportCallbackCreateInfoEXT} is:
```c
typedef struct VkDebugReportCallbackCreateInfoEXT {
    VkStructureType sType;
    const void* pNext;
    VkDebugReportFlagsEXT flags;
    PFN_vkDebugReportCallbackEXT pfnCallback;
    void* pUserData;
} VkDebugReportCallbackCreateInfoEXT;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to a structure extending this structure.
- **flags** is a bitmask of `VkDebugReportFlagBitsEXT` specifying which event(s) will cause this callback to be called.
- **pfnCallback** is the application callback function to call.
- **pUserData** is user data to be passed to the callback.

For each `VkDebugReportCallbackEXT` that is created the `VkDebugReportCallbackCreateInfoEXT::flags` determine when that `VkDebugReportCallbackCreateInfoEXT::pfnCallback` is called. When an event happens, the implementation will do a bitwise AND of the event's `VkDebugReportFlagBitsEXT` flags to each `VkDebugReportCallbackEXT` object's flags. For each non-zero result the corresponding callback will be called. The callback will come directly from the component that detected the event, unless some other layer intercepts the calls for its own purposes (filter them in a different way, log to a system error log, etc.).

An application may receive multiple callbacks if multiple `VkDebugReportCallbackEXT` objects were created. A callback will always be executed in the same thread as the originating Vulkan call.

A callback may be called from multiple threads simultaneously (if the application is making Vulkan calls from multiple threads).

---

**Valid Usage (Implicit)**

- **VUID-VkDebugReportCallbackCreateInfoEXT-sType-sType**
  - **sType** must be `VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT`

- **VUID-VkDebugReportCallbackCreateInfoEXT-flags-parameter**
  - **flags** must be a valid combination of `VkDebugReportFlagBitsEXT` values

- **VUID-VkDebugReportCallbackCreateInfoEXT-pfnCallback-parameter**
  - **pfnCallback** must be a valid `PFN_vkDebugReportCallbackEXT` value

Bits which can be set in `VkDebugReportCallbackCreateInfoEXT::flags`, specifying events which cause a debug report, are:
typedef enum VkDebugReportFlagBitsEXT {
    VK_DEBUG_REPORT_INFORMATION_BIT_EXT = 0x00000001,
    VK_DEBUG_REPORT_WARNING_BIT_EXT = 0x00000002,
    VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT = 0x00000004,
    VK_DEBUG_REPORT_ERROR_BIT_EXT = 0x00000008,
    VK_DEBUG_REPORT_DEBUG_BIT_EXT = 0x00000010,
} VkDebugReportFlagBitsEXT;

- **VK_DEBUG_REPORT_ERROR_BIT_EXT** specifies that the application has violated a valid usage condition of the specification.

- **VK_DEBUG_REPORT_WARNING_BIT_EXT** specifies use of Vulkan that may expose an app bug. Such cases may not be immediately harmful, such as a fragment shader outputting to a location with no attachment. Other cases may point to behavior that is almost certainly bad when unintended such as using an image whose memory has not been filled. In general if you see a warning but you know that the behavior is intended/desired, then simply ignore the warning.

- **VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT** specifies a potentially non-optimal use of Vulkan, e.g. using `vkCmdClearColorImage` when setting `VkAttachmentDescription::loadOp` to `VK_ATTACHMENT_LOAD_OP_CLEAR` would have worked.

- **VK_DEBUG_REPORT_INFORMATION_BIT_EXT** specifies an informational message such as resource details that may be handy when debugging an application.

- **VK_DEBUG_REPORT_DEBUG_BIT_EXT** specifies diagnostic information from the implementation and layers.

// Provided by VK_EXT_debug_report
typedef VkFlags VkDebugReportFlagsEXT;

VkDebugReportFlagsEXT is a bitmask type for setting a mask of zero or more VkDebugReportFlagBitsEXT.

The prototype for the `VkDebugReportCallbackCreateInfoEXT::PFN_vkDebugReportCallback` function implemented by the application is:

```c
#include <vkext.h>

typedef VkBool32 (VKAPI_PTR *PFN_vkDebugReportCallbackEXT)(
    VkDebugReportFlagsEXT flags,
    VkDebugReportObjectTypeEXT objectType,
    uint64_t object,
    size_t location,
    int32_t messageCode,
    const char* pLayerPrefix,
    const char* pMessage,
    void* pUserData);
```

- **flags** specifies the `VkDebugReportFlagBitsEXT` that triggered this callback.
• **objectType** is a `VkDebugReportObjectTypeEXT` value specifying the type of object being used or created at the time the event was triggered.

• **object** is the object where the issue was detected. If **objectType** is `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT`, **object** is undefined.

• **location** is a component (layer, driver, loader) defined value specifying the **location** of the trigger. This is an **optional** value.

• **messageCode** is a layer-defined value indicating what test triggered this callback.

• **pLayerPrefix** is a null-terminated string that is an abbreviation of the name of the component making the callback. **pLayerPrefix** is only valid for the duration of the callback.

• **pMessage** is a null-terminated string detailing the trigger conditions. **pMessage** is only valid for the duration of the callback.

• **pUserData** is the user data given when the `VkDebugReportCallbackEXT` was created.

The callback must not call `vkDestroyDebugReportCallbackEXT`.

The callback returns a `VkBool32`, which is interpreted in a layer-specified manner. The application should always return `VK_FALSE`. The `VK_TRUE` value is reserved for use in layer development.

**object** must be a Vulkan object or `VK_NULL_HANDLE`. If **objectType** is not `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT` and **object** is not `VK_NULL_HANDLE`, **object** must be a Vulkan object of the corresponding type associated with **objectType** as defined in `VkDebugReportObjectTypeEXT` and Vulkan Handle Relationship.

Possible values passed to the **objectType** parameter of the callback function specified by `VkDebugReportCallbackCreateInfoEXT::pfnCallback`, specifying the type of object handle being reported, are:

```c
// Provided by VK_EXT_debug_report, VK_EXT_debug_marker
typedef enum VkDebugReportObjectTypeEXT {
    VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT = 0,
    VK_DEBUG_REPORT_OBJECT_TYPE_INSTANCE_EXT = 1,
    VK_DEBUG_REPORT_OBJECT_TYPE_PHYSICAL_DEVICE_EXT = 2,
    VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT = 3,
    VK_DEBUG_REPORT_OBJECT_TYPE_QUEUE_EXT = 4,
    VK_DEBUG_REPORT_OBJECT_TYPE_SEMAPHORE_EXT = 5,
    VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_BUFFER_EXT = 6,
    VK_DEBUG_REPORT_OBJECT_TYPE_FENCE_EXT = 7,
    VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_MEMORY_EXT = 8,
    VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT = 9,
    VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT = 10,
    VK_DEBUG_REPORT_OBJECT_TYPE_EVENT_EXT = 11,
    VK_DEBUG_REPORT_OBJECT_TYPE_QUERY_POOL_EXT = 12,
    VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_VIEW_EXT = 13,
    VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_VIEW_EXT = 14,
    VK_DEBUG_REPORT_OBJECT_TYPE_SHADER_MODULE_EXT = 15,
    VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_CACHE_EXT = 16,
    VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_LAYOUT_EXT = 17,
};
```
VK_DEBUG_REPORT_OBJECT_TYPE_RENDER_PASS_EXT = 18,
VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_EXT = 19,
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT_EXT = 20,
VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_EXT = 21,
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_POOL_EXT = 22,
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_EXT = 23,
VK_DEBUG_REPORT_OBJECT_TYPE_FRAMEBUFFER_EXT = 24,
VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_POOL_EXT = 25,
VK_DEBUG_REPORT_OBJECT_TYPE_SURFACE_KHR_EXT = 26,
VK_DEBUG_REPORT_OBJECT_TYPE_SWAPCHAIN_KHR_EXT = 27,
VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT = 28,
VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_KHR_EXT = 29,
VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_MODE_KHR_EXT = 30,
VK_DEBUG_REPORT_OBJECT_TYPE_VALIDATION_CACHE_EXT_EXT = 33,
// Provided by VK_KHR_sampler_ycbcr_conversion with VK_EXT_debug_report,
VK_EXT_debug_report with VK_VERSION_1_1
VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_EXT = 1000156000,
// Provided by VK_EXT_debug_report with VK_VERSION_1_1
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT = 1000085000,
// Provided by VK_KHR_acceleration_structure
VK_DEBUG_REPORT_OBJECT_TYPE_ACCELERATION_STRUCTURE_KHR_EXT = 1000150000,
// Provided by VK_NV_ray_tracing
VK_DEBUG_REPORT_OBJECT_TYPE_ACCELERATION_STRUCTURE_NV_EXT = 1000165000,
VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_EXT =
VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT,
VK_DEBUG_REPORT_OBJECT_TYPE_VALIDATION_CACHE_EXT =
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT,
// Provided by VK_KHR_descriptor_update_template with VK_EXT_debug_report
VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR_EXT =
VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_KHR_EXT =
} VkDebugReportObjectTypeEXT;

Table 82. VkDebugReportObjectTypeEXT and Vulkan Handle Relationship

<table>
<thead>
<tr>
<th>VkDebugReportObjectTypeEXT</th>
<th>Vulkan Handle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT</td>
<td>Unknown/Undefined Handle</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_INSTANCE_EXT</td>
<td>VkInstance</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_PHYSICAL_DEVICE_EXT</td>
<td>VkPhysicalDevice</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_EXT</td>
<td>VkDevice</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_QUEUE_EXT</td>
<td>VkQueue</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_SEMAPHORE_EXT</td>
<td>VkSemaphore</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_BUFFER_EXT</td>
<td>VkCommandBuffer</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_FENCE_EXT</td>
<td>VkFence</td>
</tr>
<tr>
<td>VkDebugReportObjectTypeEXT</td>
<td>Vulkan Handle Type</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DEVICE_MEMORY_EXT</td>
<td>VkDeviceMemory</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_EXT</td>
<td>VkBuffer</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT</td>
<td>VkImage</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_EVENT_EXT</td>
<td>VkEvent</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_QUERY_POOL_EXT</td>
<td>VkQueryPool</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_BUFFER_VIEW_EXT</td>
<td>VkBufferView</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_VIEW_EXT</td>
<td>VkImageView</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_SHADER_MODULE_EXT</td>
<td>VkShaderModule</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_CACHE_EXT</td>
<td>VkPipelineCache</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_LAYOUT_EXT</td>
<td>VkPipelineLayout</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_RENDER_PASS_EXT</td>
<td>VkRenderPass</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_PIPELINE_EXT</td>
<td>VkPipeline</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT_EXT</td>
<td>VkDescriptorSetLayout</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_EXT</td>
<td>VkSampler</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_POOL_EXT</td>
<td>VkDescriptorPool</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_SET_EXT</td>
<td>VkDescriptorSet</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_FRAMEBUFFER_EXT</td>
<td>VkFramebuffer</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_COMMAND_POOL_EXT</td>
<td>VkCommandPool</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_SURFACE_KHR_EXT</td>
<td>VkSurfaceKHR</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_SWAPCHAIN_KHR_EXT</td>
<td>VkSwapchainKHR</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT</td>
<td>VkDebugReportCallbackEXT</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_KHR_EXT</td>
<td>VkDisplayKHR</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DISPLAY_MODE_KHR_EXT</td>
<td>VkDisplayModeKHR</td>
</tr>
<tr>
<td>VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT</td>
<td>VkDescriptorUpdateTemplate</td>
</tr>
</tbody>
</table>

**Note**

The primary expected use of `VK_ERROR_VALIDATION_FAILED_EXT` is for validation layer testing. It is not expected that an application would see this error code during normal use of the validation layers.
To inject its own messages into the debug stream, call:

```c
// Provided by VK_EXT_debug_report
void vkDebugReportMessageEXT(
    VkInstance                                  instance,  // Provided by VK_EXT_debug_report
    VkDebugReportFlagsEXT                       flags,      // Provided by VK_EXT_debug_report
    VkDebugReportObjectTypeEXT                  objectType, // Provided by VK_EXT_debug_report
    uint64_t                                    object,     // Provided by VK_EXT_debug_report
    int32_t                                     location,   // Provided by VK_EXT_debug_report
    size_t                                      messageCode, // Provided by VK_EXT_debug_report
    const char*                                 pLayerPrefix, // Provided by VK_EXT_debug_report
    const char*                                 pMessage);  // Provided by VK_EXT_debug_report
```

- `instance` is the debug stream's `VkInstance`.
- `flags` specifies the `VkDebugReportFlagBitsEXT` classification of this event/message.
- `objectType` is a `VkDebugReportObjectTypeEXT` specifying the type of object being used or created at the time the event was triggered.
- `object` is the object where the issue was detected. `object` can be `VK_NULL_HANDLE` if there is no object associated with the event.
- `location` is an application defined value.
- `messageCode` is an application defined value.
- `pLayerPrefix` is the abbreviation of the component making this event/message.
- `pMessage` is a null-terminated string detailing the trigger conditions.

The call will propagate through the layers and generate callback(s) as indicated by the message's flags. The parameters are passed on to the callback in addition to the `pUserData` value that was defined at the time the callback was registered.

---

**Valid Usage**

- VUID-vkDebugReportMessageEXT-object-01241
  - `object` must be a Vulkan object or `VK_NULL_HANDLE`

- VUID-vkDebugReportMessageEXT-objectType-01498
  - If `objectType` is not `VK_DEBUG_REPORT_OBJECT_TYPE_UNKNOWN_EXT` and `object` is not `VK_NULL_HANDLE`, `object` must be a Vulkan object of the corresponding type associated with `objectType` as defined in `VkDebugReportObjectTypeEXT` and Vulkan Handle Relationship
Valid Usage (Implicit)

- VUID-vkDebugReportMessageEXT-instance-parameter
  instance must be a valid VkInstance handle

- VUID-vkDebugReportMessageEXT-flags-parameter
  flags must be a valid combination of VkDebugReportFlagBitsEXT values

- VUID-vkDebugReportMessageEXT-flags-requiredbitmask
  flags must not be 0

- VUID-vkDebugReportMessageEXT-objectType-parameter
  objectType must be a valid VkDebugReportObjectTypeEXT value

- VUID-vkDebugReportMessageEXT-pLayerPrefix-parameter
  pLayerPrefix must be a null-terminated UTF-8 string

- VUID-vkDebugReportMessageEXT-pMessage-parameter
  pMessage must be a null-terminated UTF-8 string

To destroy a VkDebugReportCallbackEXT object, call:

```c
// Provided by VK_EXT_debug_report
void vkDestroyDebugReportCallbackEXT(  
    VkInstance                      instance,  
    VkDebugReportCallbackEXT       callback,  
    const VkAllocationCallbacks*  pAllocator);
```

- instance is the instance where the callback was created.
- callback is the VkDebugReportCallbackEXT object to destroy. callback is an externally synchronized object and must not be used on more than one thread at a time. This means that vkDestroyDebugReportCallbackEXT must not be called when a callback is active.
- pAllocator controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- VUID-vkDestroyDebugReportCallbackEXT-instance-01242
  If VkAllocationCallbacks were provided when callback was created, a compatible set of callbacks must be provided here

- VUID-vkDestroyDebugReportCallbackEXT-instance-01243
  If no VkAllocationCallbacks were provided when callback was created, pAllocator must be NULL
Valid Usage (Implicit)

- **VUID-vkDestroyDebugReportCallbackEXT-instance-parameter**
  
  *instance* must be a valid *VkInstance* handle

- **VUID-vkDestroyDebugReportCallbackEXT-callback-parameter**
  
  If *callback* is not *VK_NULL_HANDLE*, *callback* must be a valid *VkDebugReportCallbackEXT* handle

- **VUID-vkDestroyDebugReportCallbackEXT-pAllocator-parameter**
  
  If *pAllocator* is not NULL, *pAllocator* must be a valid pointer to a valid *VkAllocationCallbacks* structure

- **VUID-vkDestroyDebugReportCallbackEXT-callback-parent**
  
  If *callback* is a valid handle, it must have been created, allocated, or retrieved from *instance*

---

Host Synchronization

- Host access to *callback* must be externally synchronized

---

45.4. Device Loss Debugging

45.4.1. Device Diagnostic Checkpoints

Device execution progress can be tracked for the purposes of debugging a device loss by annotating the command stream with application-defined diagnostic checkpoints.

Device diagnostic checkpoints are inserted into the command stream by calling *vkCmdSetCheckpointNV*.

```c
// Provided by VK_NV_device_diagnostic_checkpoints
void vkCmdSetCheckpointNV(
    VkCommandBuffer commandBuffer,       // Provided by VK_NV_device_diagnostic_checkpoints
    const void* pCheckpointMarker);     // Provided by VK_NV_device_diagnostic_checkpoints
```

- *commandBuffer* is the command buffer that will receive the marker
- *pCheckpointMarker* is an opaque application-provided value that will be associated with the checkpoint.
Valid Usage (Implicit)

- VUID-vkCmdSetCheckpointNV-commandBuffer-parameter  
  `commandBuffer` must be a valid `VkCommandBuffer` handle

- VUID-vkCmdSetCheckpointNV-commandBuffer-recording  
  `commandBuffer` must be in the recording state

- VUID-vkCmdSetCheckpointNV-commandBuffer-cmdpool  
  The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, compute, or transfer operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

- Host access to the `VkCommandPool` that `commandBuffer` was allocated from must be externally synchronized

Command Properties

<table>
<thead>
<tr>
<th>Command Buffer Levels</th>
<th>Render Pass Scope</th>
<th>Supported Queue Types</th>
<th>Pipeline Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transfer</td>
<td></td>
</tr>
</tbody>
</table>

Note that `pCheckpointMarker` is treated as an opaque value. It does not need to be a valid pointer and will not be dereferenced by the implementation.

If the device encounters an error during execution, the implementation will return a `VK_ERROR_DEVICE_LOST` error to the application at some point during host execution. When this happens, the application can call `vkGetQueueCheckpointData2NV` to retrieve information on the most recent diagnostic checkpoints that were executed by the device.

```c
// Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
void vkGetQueueCheckpointData2NV(
    VkQueue queue,
    uint32_t* pCheckpointDataCount,
    VkCheckpointData2NV* pCheckpointData);
```

- `queue` is the `VkQueue` object the caller would like to retrieve checkpoint data for
- `pCheckpointDataCount` is a pointer to an integer related to the number of checkpoint markers available or queried, as described below.
• pCheckpointData is either NULL or a pointer to an array of VkCheckpointData2NV structures.

If pCheckpointData is NULL, then the number of checkpoint markers available is returned in pCheckpointDataCount. Otherwise, pCheckpointDataCount must point to a variable set by the user to the number of elements in the pCheckpointData array, and on return the variable is overwritten with the number of structures actually written to pCheckpointData.

If pCheckpointDataCount is less than the number of checkpoint markers available, at most pCheckpointDataCount structures will be written.

Valid Usage

• VUID-vkGetQueueCheckpointData2NV-queue-03892
  The device that queue belongs to must be in the lost state

Valid Usage (Implicit)

• VUID-vkGetQueueCheckpointData2NV-queue-parameter
  queue must be a valid VkQueue handle

• VUID-vkGetQueueCheckpointData2NV-pCheckpointDataCount-parameter
  pCheckpointDataCount must be a valid pointer to a uint32_t value

• VUID-vkGetQueueCheckpointData2NV-pCheckpointData-parameter
  If the value referenced by pCheckpointDataCount is not 0, and pCheckpointData is not NULL, pCheckpointData must be a valid pointer to an array of pCheckpointDataCount VkCheckpointData2NV structures

The VkCheckpointData2NV structure is defined as:

```c
// Provided by VK_KHR_synchronization2 with VK_NV_device_diagnostic_checkpoints
typedef struct VkCheckpointData2NV {
    VkStructureType             sType;         // sType is the type of this structure.
    void*                       pNext;         // pNext is NULL or a pointer to a structure extending this structure.
    VkPipelineStageFlags2KHR    stage;         // stage indicates a single pipeline stage which the checkpoint marker data refers to.
    void*                       pCheckpointMarker; // pCheckpointMarker contains the value of the last checkpoint marker executed in the stage that stage refers to.
} VkCheckpointData2NV;
```
Valid Usage (Implicit)

- **VUID-VkCheckpointData2NV-sType-sType**
  - `sType` **must** be `VK_STRUCTURE_TYPE_CHECKPOINT_DATA_2_NV`

- **VUID-VkCheckpointData2NV-pNext-pNext**
  - `pNext` **must** be `NULL`

The stages at which a checkpoint marker **can** be executed are implementation-defined and **can** be queried by calling `vkGetPhysicalDeviceQueueFamilyProperties2`.

If the device encounters an error during execution, the implementation will return a `VK_ERROR_DEVICE_LOST` error to the application at a certain point during host execution. When this happens, the application **can** call `vkGetQueueCheckpointDataNV` to retrieve information on the most recent diagnostic checkpoints that were executed by the device.

```c
// Provided by VK_NV_device_diagnostic_checkpoints
void vkGetQueueCheckpointDataNV(
    VkQueue                                     queue,
    uint32_t*                                   pCheckpointDataCount,  // number of checkpoint markers available or queried
    VkCheckpointDataNV*                         pCheckpointData);         // number of elements in the pCheckpointData array, and on return the variable is overwritten with the number of structures actually written
```

- `queue` is the `VkQueue` object the caller would like to retrieve checkpoint data for

- `pCheckpointDataCount` is a pointer to an integer related to the number of checkpoint markers available or queried, as described below.

- `pCheckpointData` is either `NULL` or a pointer to an array of `VkCheckpointDataNV` structures.

If `pCheckpointData` is `NULL`, then the number of checkpoint markers available is returned in `pCheckpointDataCount`.

Otherwise, `pCheckpointDataCount` **must** point to a variable set by the user to the number of elements in the `pCheckpointData` array, and on return the variable is overwritten with the number of structures actually written to `pCheckpointData`.

If `pCheckpointDataCount` is less than the number of checkpoint markers available, at most `pCheckpointDataCount` structures will be written.

Valid Usage

- **VUID-vkGetQueueCheckpointDataNV-queue-02025**
  - The device that `queue` belongs to **must** be in the lost state
Valid Usage (Implicit)

- VUID-vkGetQueueCheckpointDataNV-queue-parameter
  
  queue must be a valid VkQueue handle

- VUID-vkGetQueueCheckpointDataNV-pCheckpointDataCount-parameter
  
  pCheckpointDataCount must be a valid pointer to a uint32_t value

- VUID-vkGetQueueCheckpointDataNV-pCheckpointData-parameter
  
  If the value referenced by pCheckpointDataCount is not 0, and pCheckpointData is not NULL, pCheckpointData must be a valid pointer to an array of pCheckpointDataCount VkCheckpointDataNV structures

The VkCheckpointDataNV structure is defined as:

```
// Provided by VK_NV_device_diagnostic_checkpoints
typedef struct VkCheckpointDataNV {
    VkStructureType sType;
    void* pNext;
    VkPipelineStageFlagBits stage;
    void* pCheckpointMarker;
} VkCheckpointDataNV;
```

- sType is the type of this structure.
- pNext is NULL or a pointer to a structure extending this structure.
- stage indicates which pipeline stage the checkpoint marker data refers to.
- pCheckpointMarker contains the value of the last checkpoint marker executed in the stage that stage refers to.

The stages at which a checkpoint marker can be executed are implementation-defined and can be queried by calling vkGetPhysicalDeviceQueueFamilyProperties2.

Valid Usage (Implicit)

- VUID-VkCheckpointDataNV-sType-sType
  
  sType must be VK_STRUCTURE_TYPE_CHECKPOINT_DATA_NV

- VUID-VkCheckpointDataNV-pNext-pNext
  
  pNext must be NULL

45.5. Active Tooling Information

Information about tools providing debugging, profiling, or similar services, active for a given physical device, can be obtained by calling:
// Provided by VK_EXT_tooling_info
VkResult vkGetPhysicalDeviceToolPropertiesEXT(
    VkPhysicalDevice                            physicalDevice,
    uint32_t*                                   pToolCount,
    VkPhysicalDeviceToolPropertiesEXT*          pToolProperties);

- `physicalDevice` is the handle to the physical device to query for active tools.
- `pToolCount` is a pointer to an integer describing the number of tools active on `physicalDevice`.
- `pToolProperties` is either `NULL` or a pointer to an array of `VkPhysicalDeviceToolPropertiesEXT` structures.

If `pToolProperties` is `NULL`, then the number of tools currently active on `physicalDevice` is returned in `pToolCount`. Otherwise, `pToolCount` must point to a variable set by the user to the number of elements in the `pToolProperties` array, and on return the variable is overwritten with the number of structures actually written to `pToolProperties`. If `pToolCount` is less than the number of currently active tools, at most `pToolCount` structures will be written.

The count and properties of active tools may change in response to events outside the scope of the specification. An application should assume these properties might change at any given time.

**Valid Usage (Implicit)**

- VUID-vkGetPhysicalDeviceToolPropertiesEXT-physicalDevice-parameter
  `physicalDevice` must be a valid `VkPhysicalDevice` handle

- VUID-vkGetPhysicalDeviceToolPropertiesEXT-pToolCount-parameter
  `pToolCount` must be a valid pointer to a `uint32_t` value

- VUID-vkGetPhysicalDeviceToolPropertiesEXT-pToolProperties-parameter
  If the value referenced by `pToolCount` is not 0, and `pToolProperties` is not NULL, `pToolProperties` must be a valid pointer to an array of `pToolCount` `VkPhysicalDeviceToolPropertiesEXT` structures

**Return Codes**

**Success**

- VK_SUCCESS
- VK_INCOMPLETE

**Failure**

- VK_ERROR_OUT_OF_HOST_MEMORY

The `VkPhysicalDeviceToolPropertiesEXT` structure is defined as:
```c
// Provided by VK_EXT_tooling_info
typedef struct VkPhysicalDeviceToolPropertiesEXT {
    VkStructureType          sType;
    void*                    pNext;
    char                     name[VK_MAX_EXTENSION_NAME_SIZE];
    char                     version[VK_MAX_EXTENSION_NAME_SIZE];
    VkToolPurposeFlagsEXT    purposes;
    char                     description[VK_MAX_DESCRIPTION_SIZE];
    char                     layer[VK_MAX_EXTENSION_NAME_SIZE];
} VkPhysicalDeviceToolPropertiesEXT;
```

- **sType** is the type of this structure.
- **pNext** is **NULL** or a pointer to a structure extending this structure.
- **name** is a null-terminated UTF-8 string containing the name of the tool.
- **version** is a null-terminated UTF-8 string containing the version of the tool.
- **purposes** is a bitmask of `VkToolPurposeFlagBitsEXT` which is populated with purposes supported by the tool.
- **description** is a null-terminated UTF-8 string containing a description of the tool.
- **layer** is a null-terminated UTF-8 string that contains the name of the layer implementing the tool, if the tool is implemented in a layer - otherwise it **may** be an empty string.

### Valid Usage (Implicit)

- **VUID-VkPhysicalDeviceToolPropertiesEXT-sType-sType**
  - **sType** must be `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TOOL_PROPERTIES_EXT`
- **VUID-VkPhysicalDeviceToolPropertiesEXT-pNext-pNext**
  - **pNext** must be **NULL**

Bits which **can** be set in `VkDeviceQueueCreateInfo::purposes` specifying the purposes of an active tool are:
typedef enum VkToolPurposeFlagBitsEXT {
    VK_TOOL_PURPOSE_VALIDATION_BIT_EXT = 0x00000001,
    VK_TOOL_PURPOSE_PROFILING_BIT_EXT = 0x00000002,
    VK_TOOL_PURPOSE_TRACING_BIT_EXT = 0x00000004,
    VK_TOOL_PURPOSE_ADDITIONAL_FEATURES_BIT_EXT = 0x00000008,
    VK_TOOL_PURPOSE_MODIFYING_FEATURES_BIT_EXT = 0x00000010,
    // Provided by VK_EXT_tooling_info with VK_EXT_debug_report, VK_EXT_tooling_info
    with VK_EXT_debug_utils
    VK_TOOL_PURPOSE_DEBUG_REPORTING_BIT_EXT = 0x00000020,
    // Provided by VK_EXT_tooling_info with VK_EXT_debug_marker, VK_EXT_tooling_info
    with VK_EXT_debug_utils
    VK_TOOL_PURPOSE_DEBUG_MARKERS_BIT_EXT = 0x00000040,
} VkToolPurposeFlagBitsEXT;

• VK_TOOL_PURPOSE_VALIDATION_BIT_EXT specifies that the tool provides validation of API usage.
• VK_TOOL_PURPOSE_PROFILING_BIT_EXT specifies that the tool provides profiling of API usage.
• VK_TOOL_PURPOSE_TRACING_BIT_EXT specifies that the tool is capturing data about the application’s API usage, including anything from simple logging to capturing data for later replay.
• VK_TOOL_PURPOSE_ADDITIONAL_FEATURES_BIT_EXT specifies that the tool provides additional API features/extensions on top of the underlying implementation.
• VK_TOOL_PURPOSE_MODIFYING_FEATURES_BIT_EXT specifies that the tool modifies the API features/limits/extensions presented to the application.
• VK_TOOL_PURPOSE_DEBUG_REPORTING_BIT_EXT specifies that the tool reports additional information to the application via callbacks specified by vkCreateDebugReportCallbackEXT or vkCreateDebugUtilsMessengerEXT
• VK_TOOL_PURPOSE_DEBUG_MARKERS_BIT_EXT specifies that the tool consumes debug markers or object debug annotation, queue labels, or command buffer labels

typedef VkFlags VkToolPurposeFlagsEXT;

VkToolPurposeFlagsEXT is a bitmask type for setting a mask of zero or more VkToolPurposeFlagBitsEXT.
Appendix A: Vulkan Environment for SPIR-V

Shaders for Vulkan are defined by the Khronos SPIR-V Specification as well as the Khronos SPIR-V Extended Instructions for GLSL Specification. This appendix defines additional SPIR-V requirements applying to Vulkan shaders.

Versions and Formats

A Vulkan 1.1 implementation must support the 1.0, 1.1, 1.2, and 1.3 versions of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL. If the VK_KHR_spirv_1_4 extension is enabled, the implementation must additionally support the 1.4 version of SPIR-V.

A SPIR-V module passed into vkCreateShaderModule is interpreted as a series of 32-bit words in host endianness, with literal strings packed as described in section 2.2 of the SPIR-V Specification. The first few words of the SPIR-V module must be a magic number and a SPIR-V version number, as described in section 2.3 of the SPIR-V Specification.

Capabilities

The table below lists the set of SPIR-V capabilities that may be supported in Vulkan implementations. The application must not use any of these capabilities in SPIR-V passed to vkCreateShaderModule unless one of the following conditions is met for the VkDevice specified in the device parameter of vkCreateShaderModule:

- The corresponding field in the table is blank.
- Any corresponding Vulkan feature is enabled.
- Any corresponding Vulkan extension is enabled.
- Any corresponding Vulkan property is supported.
- The corresponding core version is supported (as returned by VkPhysicalDeviceProperties::apiVersion).

Table 83. List of SPIR-V Capabilities and corresponding Vulkan features, extensions, or core version

<table>
<thead>
<tr>
<th>SPIR-V OpCapability</th>
<th>Vulkan feature, extension, or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>Shader</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>InputAttachment</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>Sampled1D</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
<td>Vulkan feature, extension, or core version</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Image1D</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>SampledBuffer</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>ImageBuffer</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>ImageQuery</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>DerivativeControl</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>Geometry</td>
<td>VkPhysicalDeviceFeatures::geometryShader</td>
</tr>
<tr>
<td>Tessellation</td>
<td>VkPhysicalDeviceFeatures::tessellationShader</td>
</tr>
<tr>
<td>Float64</td>
<td>VkPhysicalDeviceFeatures::shaderFloat64</td>
</tr>
<tr>
<td>Int64</td>
<td>VkPhysicalDeviceFeatures::shaderInt64</td>
</tr>
<tr>
<td>Int64Atomics</td>
<td>VkPhysicalDeviceVulkan12Features::shaderBufferInt64Atomics</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceVulkan12Features::shaderSharedInt64Atomics</td>
</tr>
<tr>
<td>AtomicFloat32AddEXT</td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderBufferFloat32AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderSharedFloat32AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderImageFloat32AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::sparseImageFloat32AtomicAdd</td>
</tr>
<tr>
<td>AtomicFloat64AddEXT</td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderBufferFloat64AtomicAdd</td>
</tr>
<tr>
<td></td>
<td>VkPhysicalDeviceShaderAtomicFloatFeaturesEXT::shaderSharedFloat64AtomicAdd</td>
</tr>
<tr>
<td>Int64ImageEXT</td>
<td>VkPhysicalDeviceShaderImageAtomicInt64FeaturesEXT::shaderImageInt64Atomics</td>
</tr>
<tr>
<td>Int16</td>
<td>VkPhysicalDeviceFeatures::shaderInt16</td>
</tr>
<tr>
<td>TessellationPointSize</td>
<td>VkPhysicalDeviceFeatures::shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>GeometryPointSize</td>
<td>VkPhysicalDeviceFeatures::shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>ImageGatherExtended</td>
<td>VkPhysicalDeviceFeatures::shaderImageGatherExtended</td>
</tr>
<tr>
<td>SPIR-V</td>
<td>OpCapability</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>Vulkan feature, extension, or core version</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StorageImageMultisample</th>
<th>VkPhysicalDeviceFeatures::shaderStorageImageMultisample</th>
</tr>
</thead>
<tbody>
<tr>
<td>UniformBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderUniformBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>SampledImageArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderSampledImageArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageBufferArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderStorageBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>StorageImageArrayDynamicIndexing</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageArrayDynamicIndexing</td>
</tr>
<tr>
<td>ClipDistance</td>
<td>VkPhysicalDeviceFeatures::shaderClipDistance</td>
</tr>
<tr>
<td>CullDistance</td>
<td>VkPhysicalDeviceFeatures::shaderCullDistance</td>
</tr>
<tr>
<td>ImageCubeArray</td>
<td>VkPhysicalDeviceFeatures::imageCubeArray</td>
</tr>
<tr>
<td>SampleRateShading</td>
<td>VkPhysicalDeviceFeatures::sampleRateShading</td>
</tr>
<tr>
<td>SparseResidency</td>
<td>VkPhysicalDeviceFeatures::shaderResourceResidency</td>
</tr>
<tr>
<td>MinLod</td>
<td>VkPhysicalDeviceFeatures::shaderResourceMinLod</td>
</tr>
<tr>
<td>SampledCubeArray</td>
<td>VkPhysicalDeviceFeatures::imageCubeArray</td>
</tr>
<tr>
<td>ImageMSArray</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageMultisample</td>
</tr>
<tr>
<td>StorageImageExtendedFormats</td>
<td>VK_API_VERSION_1_0</td>
</tr>
<tr>
<td>InterpolationFunction</td>
<td>VkPhysicalDeviceFeatures::sampleRateShading</td>
</tr>
<tr>
<td>StorageImageReadWithoutFormat</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageReadWithoutFormat</td>
</tr>
<tr>
<td>StorageImageWriteWithoutFormat</td>
<td>VkPhysicalDeviceFeatures::shaderStorageImageWriteWithoutFormat</td>
</tr>
<tr>
<td>MultiViewport</td>
<td>VkPhysicalDeviceFeatures::multiViewport</td>
</tr>
<tr>
<td>Feature</td>
<td>Vulkan Extension</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
<td>Vulkan feature, extension, or core version</td>
</tr>
<tr>
<td>DrawParameters</td>
<td>VkPhysicalDeviceShaderDrawParametersFeatures::shaderDrawParameters</td>
</tr>
<tr>
<td></td>
<td>VK_KHR_shader_draw_parameters</td>
</tr>
<tr>
<td>MultiView</td>
<td>VkPhysicalDeviceMultiviewFeatures::multiview</td>
</tr>
<tr>
<td>DeviceGroup</td>
<td>VK_API_VERSION_1_1</td>
</tr>
<tr>
<td></td>
<td>VK_KHR_device_group</td>
</tr>
<tr>
<td>VariablePointersStorageBuffer</td>
<td>VkPhysicalDeviceVariablePointersFeatures::variablePointersStorageBuffer</td>
</tr>
<tr>
<td>VariablePointers</td>
<td>VkPhysicalDeviceVariablePointersFeatures::variablePointers</td>
</tr>
<tr>
<td>ShaderClockKHR</td>
<td>VK_KHR_shader_clock</td>
</tr>
<tr>
<td>StencilExportEXT</td>
<td>VK_EXT_shader_stencil_export</td>
</tr>
<tr>
<td>SubgroupBallotKHR</td>
<td>VK_EXT_shader_subgroup_ballot</td>
</tr>
<tr>
<td>SubgroupVoteKHR</td>
<td>VK_EXT_shader_subgroup_vote</td>
</tr>
<tr>
<td>ImageReadWriteLodAMD</td>
<td>VK_AMD_shader_image_load_store_lod</td>
</tr>
<tr>
<td>ImageGatherBiasLodAMD</td>
<td>VK_AMD_texture_gather_bias_lod</td>
</tr>
<tr>
<td>FragmentMaskAMD</td>
<td>VK_AMD_shader_fragment_mask</td>
</tr>
<tr>
<td>SampleMaskOverrideCoverageNV</td>
<td>VK_NV_sample_mask_override_coverage</td>
</tr>
<tr>
<td>GeometryShaderPassthroughNV</td>
<td>VK_NV_geometry_shader_passthrough</td>
</tr>
<tr>
<td>ShaderViewportIndexLayerEXT</td>
<td>VK_EXT_shader_viewport_index_layer</td>
</tr>
<tr>
<td>ShaderViewportIndexLayerNV</td>
<td>VK_NV_viewport_array2</td>
</tr>
<tr>
<td>ShaderViewportMaskNV</td>
<td>VK_NV_viewport_array2</td>
</tr>
<tr>
<td>PerViewAttributesNV</td>
<td>VK_NVX_multiview_per_view_attributes</td>
</tr>
<tr>
<td>SPIR-V</td>
<td>OpCapability</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>Vulkan feature, extension, or core version</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StorageBuffer16BitAccess</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDevice16BitStorageFeatures::storageBuffer16BitAccess</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UniformAndStorageBuffer16BitAccess</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDevice16BitStorageFeatures::uniformAndStorageBuffer16BitAccess</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StoragePushConstant16</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDevice16BitStorageFeatures::storagePushConstant16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StorageInputOutput16</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDevice16BitStorageFeatures::storageInputOutput16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_BASIC_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformVote</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_VOTE_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformArithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_ARITHMETIC_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformBallot</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_BALLOT_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformShuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_SHUFFLE_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformShuffleRelative</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_SHUFFLE_RELATIVE_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformClustered</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_CLUSTERED_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformQuad</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_QUAD_BIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GroupNonUniformPartitionedNV</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SampleMaskPostDepthCoverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXT_post_depth_coverage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ShaderNonUniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_EXT_descriptor_indexing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RuntimeDescriptorArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Features::runtimeDescriptorArray</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>InputAttachmentArrayDynamicIndexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Features::shaderInputAttachmentArrayDynamicIndexing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UniformTexelBufferArrayDynamicIndexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Features::shaderUniformTexelBufferArrayDynamicIndexing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StorageTexelBufferArrayDynamicIndexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>VkPhysicalDeviceVulkan12Features::shaderStorageTexelBufferArrayDynamicIndexing</td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>UniformBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>SampledImageArrayNonUniformIndexing</td>
</tr>
<tr>
<td>StorageBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>StorageImageArrayNonUniformIndexing</td>
</tr>
<tr>
<td>InputAttachmentArrayNonUniformIndexing</td>
</tr>
<tr>
<td>UniformTexelBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>StorageTexelBufferArrayNonUniformIndexing</td>
</tr>
<tr>
<td>Float16</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Int8</td>
</tr>
<tr>
<td>StorageBuffer8BitAccess</td>
</tr>
<tr>
<td>UniformAndStorageBuffer8BitAccess</td>
</tr>
<tr>
<td>StoragePushConstant8</td>
</tr>
<tr>
<td>VulkanMemoryModel</td>
</tr>
<tr>
<td>VulkanMemoryModelDeviceScope</td>
</tr>
<tr>
<td>DenormPreserve</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>DenormFlushToZero</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SPIR-V OpCapability</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
</tbody>
</table>
| SignedZeroInfNanPreserve | VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat16  
VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat32  
VkPhysicalDeviceVulkan12Properties::shaderSignedZeroInfNanPreserveFloat64 |
| RoundingModeRTE | VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat16  
VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat32  
VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTEFloat64 |
| RoundingModeRTZ | VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat16  
VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat32  
VkPhysicalDeviceVulkan12Properties::shaderRoundingModeRTZFloat64 |
| ComputeDerivativeGroupQuadsNV | VkPhysicalDeviceComputeShaderDerivativesFeaturesNV::computeDerivativeGroupQuads |
| ComputeDerivativeGroupLinearNV | VkPhysicalDeviceComputeShaderDerivativesFeaturesNV::computeDerivativeGroupLinear |
| FragmentBarycentricNV | VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV::fragmentShaderBarycentric |
| ImageFootprintNV | VkPhysicalDeviceShaderImageFootprintFeaturesNV::imageFootprint |
| ShadingRateNV | VkPhysicalDeviceShadingRateImageFeaturesNV::shadingRateImage |
| MeshShadingNV | VK_NV_mesh_shader |
| RayTracingKHR | VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTracingPipeline |
| RayQueryKHR | VkPhysicalDeviceRayQueryFeaturesKHR::rayQuery |
| RayTraversalPrimitiveCullingKHR | VkPhysicalDeviceRayTracingPipelineFeaturesKHR::rayTraversalPrimitiveCulling |
| RayTracingNV | VK_NV_ray_tracing |
| TransformFeedback | VkPhysicalDeviceTransformFeedbackFeaturesEXT::transformFeedback |
| GeometryStreams | VkPhysicalDeviceTransformFeedbackFeaturesEXT::geometryStreams |
| FragmentDensityEXT | VkPhysicalDeviceFragmentDensityMapFeaturesEXT::fragmentDensityMap |
The application **must** not pass a SPIR-V module containing any of the following to `vkCreateShaderModule`:

- any `OpCapability` not listed above,
- an unsupported capability, or
- a capability which corresponds to a Vulkan feature or extension which has not been enabled.
SPIR-V Extensions

The following table lists SPIR-V extensions that implementations may support. The application must not pass a SPIR-V module to vkCreateShaderModule that uses the following SPIR-V extensions unless one of the following conditions is met for the VkDevice specified in the device parameter of vkCreateShaderModule:

- Any corresponding Vulkan extension is enabled.
- The corresponding core version is supported (as returned by VkPhysicalDeviceProperties::apiVersion).

Table 84. List of SPIR-V Extensions and corresponding Vulkan extensions or core version

<table>
<thead>
<tr>
<th>SPIR-V OpExtension</th>
<th>Vulkan extension or core version</th>
</tr>
</thead>
</table>
| SPV_KHR_variable_pointers | VK_API_VERSION_1_1  
VK_KHR_variable_pointers |
| SPV_AMD_shader_explicit_vertex_parameter | VK_AMD_shader_explicit_vertex_parameter |
| SPV_AMD_gcn_shader | VK_AMD_gcn_shader |
| SPV_AMD_gpu_shader_half_float | VK_AMD_gpu_shader_half_float |
| SPV_AMD_gpu_shader_int16 | VK_AMD_gpu_shader_int16 |
| SPV_AMD_shader_ballot | VK_AMD_shader_ballot |
| SPV_AMD_shader_fragment_mask | VK_AMD_shader_fragment_mask |
| SPV_AMD_shader_image_load_store_lod | VK_AMD_shader_image_load_store_lod |
| SPV_AMD_shader_trinary_minmax | VK_AMD_shader_trinary_minmax |
| SPV_AMD_texture_gather_bias_lod | VK_AMD_texture_gather_bias_lod |
| SPV_KHR_shader_draw_parameters | VK_API_VERSION_1_1  
VK_KHR_shader_draw_parameters |
<p>| SPV_KHR_8bit_storage | VK_KHR_8bit_storage |</p>
<table>
<thead>
<tr>
<th>SPIR-V 0pExtension</th>
<th>Vulkan extension or core version</th>
</tr>
</thead>
</table>
| SPV_KHR_16bit_storage | VK_API_VERSION_1_1
|                     | VK_KHR_16bit_storage |
| SPV_KHR_shader_clock | VK_KHR_shader_clock |
| SPV_KHR_float_controls | VK_KHR_shader_float_controls |
| SPV_KHR_storage_buffer_storage_class | VK_API_VERSION_1_1
<p>|                     | VK_KHR_storage_buffer_storage_class |
| SPV_KHR_post_depth_coverage | VK_EXT_post_depth_coverage |
| SPV_EXT_shader_stencil_export | VK_EXT_shader_stencil_export |
| SPV_KHR_shader_ballot | VK_EXT_shader_subgroup_ballot |
| SPV_KHR_subgroup_vote | VK_EXT_shader_subgroup_vote |
| SPV_NV_sample_mask_override_coverage | VK_NV_sample_mask_override_coverage |
| SPV_NV_geometry_shader_passthrough | VK_NV_geometry_shader_passthrough |
| SPV_NV_mesh_shader | VK_NV_mesh_shader |
| SPV_NV_viewport_array2 | VK_NV_viewport_array2 |
| SPV_NV_shader_subgroup_partitioned | VK_NV_shader_subgroup_partitioned |
| SPV_EXT_shader_viewport_index_layer | VK_EXT_shader_viewport_index_layer |
| SPV_NVX_multiview_per_view_attributes | VK_NVX_multiview_per_view_attributes |
| SPV_EXT_descriptor_indexing | VK_EXT_descriptor_indexing |
| SPV_KHR_vulkan_memory_model | VK_KHR_vulkan_memory_model |
| SPV_NV_compute_shader_derivatives | VK_NV_compute_shader_derivatives |</p>
<table>
<thead>
<tr>
<th>SPIR-V OpExtension</th>
<th>Vulkan extension or core version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPV_NV_fragment_shader_barycentric</td>
<td>VK_NV_fragment_shader_barycentric</td>
</tr>
<tr>
<td>SPV_NV_shader_image_footprint</td>
<td>VK_NV_shader_image_footprint</td>
</tr>
<tr>
<td>SPV_NV_shading_rate</td>
<td>VK_NV_shading_rate_image</td>
</tr>
<tr>
<td>SPV_NV_ray_tracing</td>
<td>VK_NV_ray_tracing</td>
</tr>
<tr>
<td>SPV_KHR_ray_tracing</td>
<td>VK_KHR_ray_tracing_pipeline</td>
</tr>
<tr>
<td>SPV_KHR_ray_query</td>
<td>VK_KHR_ray_query</td>
</tr>
<tr>
<td>SPV_GOOGLE_hlsl_functionality1</td>
<td>VK_GOOGLE_hlsl_functionality1</td>
</tr>
<tr>
<td>SPV_GOOGLE_user_type</td>
<td>VK_GOOGLE_user_type</td>
</tr>
<tr>
<td>SPV_GOOGLE_decorate_string</td>
<td>VK_GOOGLE_decorate_string</td>
</tr>
<tr>
<td>SPV_EXT_fragment_invocation_density</td>
<td>VK_EXT_fragment_density_map</td>
</tr>
<tr>
<td>SPV_KHR_physical_storage_buffer</td>
<td>VK_KHR_buffer_device_address</td>
</tr>
<tr>
<td>SPV_EXT_physical_storage_buffer</td>
<td>VK_EXT_buffer_device_address</td>
</tr>
<tr>
<td>SPV_NV_cooperative_matrix</td>
<td>VK_NV_cooperative_matrix</td>
</tr>
<tr>
<td>SPV_NV_shader_sm_builtins</td>
<td>VK_NV_shader_sm_builtins</td>
</tr>
<tr>
<td>SPV_EXT_fragment_shader_interlock</td>
<td>VK_EXT_fragment_shader_interlock</td>
</tr>
<tr>
<td>SPV_EXT_demote_to_helper_invocation</td>
<td>VK_EXT_shader_demote_to_helper_invocation</td>
</tr>
<tr>
<td>SPV_KHR_fragment_shading_rate</td>
<td>VK_KHR_fragment_shading_rate</td>
</tr>
<tr>
<td>SPV_KHR_non_semantic_info</td>
<td>VK_KHR_shader_non_semantic_info</td>
</tr>
<tr>
<td>SPV_EXT_shader_image_int64</td>
<td>VK_EXT_shader_image_atomic_int64</td>
</tr>
</tbody>
</table>
Validation Rules within a Module

A SPIR-V module passed to `vkCreateShaderModule` must conform to the following rules:

**Standalone SPIR-V Validation**

The following rules can be validated with only the SPIR-V module itself. They do not depend on knowledge of the implementation and its capabilities or knowledge of runtime information, such as enabled features.
Valid Usage

- VUID-StandaloneSpirv-None-04633
  Every entry point **must** have no return value and accept no arguments

- VUID-StandaloneSpirv-None-04634
  The static function-call graph for an entry point **must** not contain cycles; that is, static recursion is not allowed

- VUID-StandaloneSpirv-None-04635
  The Logical or PhysicalStorageBuffer64 addressing model **must** be selected

- VUID-StandaloneSpirv-None-04636
  **Scope** for execution **must** be limited to Workgroup or Subgroup

- VUID-StandaloneSpirv-None-04637
  If the **Scope** for execution is Workgroup, then it **must** only be used in the task, mesh, tessellation control, or compute execution models

- VUID-StandaloneSpirv-None-04638
  **Scope** for memory **must** be limited to Device, QueueFamily, Workgroup, ShaderCallKHR, Subgroup, or Invocation

- VUID-StandaloneSpirv-None-04639
  If the **Scope** for memory is Workgroup, then it **must** only be used in the task, mesh, or compute execution models

- VUID-StandaloneSpirv-None-04640
  If the **Scope** for memory is ShaderCallKHR, then it **must** only be used in ray generation, intersection, closest hit, any-hit, miss, and callable execution models

- VUID-StandaloneSpirv-None-04641
  If the **Scope** for memory is Invocation, then memory semantics **must** be None

- VUID-StandaloneSpirv-None-04642
  **Scope** for Non Uniform Group Operations **must** be limited to Subgroup

- VUID-StandaloneSpirv-None-04643
  **Storage Class** must be limited to UniformConstant, Input, Uniform, Output, Workgroup, Private, Function, PushConstant, Image, StorageBuffer, RayPayloadKHR, IncomingRayPayloadKHR, HitAttributeKHR, CallableDataKHR, IncomingCallableDataKHR, ShaderRecordBufferKHR, or PhysicalStorageBuffer

- VUID-StandaloneSpirv-None-04644
  If the **Storage Class** is Output, then it **must** not be used in the RayGenerationKHR, IntersectionKHR, AnyHitKHR, ClosestHitKHR, MissKHR, or CallableKHR execution models

- VUID-StandaloneSpirv-None-04645
  If the **Storage Class** is Workgroup, then it **must** only be used in the task, mesh, or compute execution models

- VUID-StandaloneSpirv-OpAtomicStore-04730
  **OpAtomicStore** **must** not use Acquire, AcquireRelease, or SequentiallyConsistent memory semantics
• **VUID-StandaloneSpirv-OpAtomicLoad-04731**  
  *OpAtomicLoad* must not use *Release, AcquireRelease*, or *SequentiallyConsistent* memory semantics.

• **VUID-StandaloneSpirv-OpMemoryBarrier-04732**  
  *OpMemoryBarrier* must use one of *Acquire, Release, AcquireRelease*, or *SequentiallyConsistent* memory semantics.

• **VUID-StandaloneSpirv-OpMemoryBarrier-04733**  
  *OpMemoryBarrier* must include at least one storage class.

• **VUID-StandaloneSpirv-OpControlBarrier-04650**  
  If the semantics for *OpControlBarrier* includes one of *Acquire, Release, AcquireRelease*, or *SequentiallyConsistent* memory semantics, then it must include at least one storage class.

• **VUID-StandaloneSpirv-OpVariable-04651**  
  Any *OpVariable* with an *Initializer* operand must have *Output, Private, Function*, or *Workgroup* as its *Storage Class* operand.

• **VUID-StandaloneSpirv-OpVariable-04734**  
  Any *OpVariable* with an *Initializer* operand and *Workgroup* as its *Storage Class* operand must use *OpConstantNull* as the initializer.

**Scope for *OpReadClockKHR*** must be limited to *Subgroup* or *Device*.

• **VUID-StandaloneSpirv-OriginLowerLeft-04653**  
  The *OriginLowerLeft* execution mode must not be used; fragment entry points must declare *OriginUpperLeft*.

• **VUID-StandaloneSpirv-PixelCenterInteger-04654**  
  The *PixelCenterInteger* execution mode must not be used (pixels are always centered at half-integer coordinates).

• **VUID-StandaloneSpirv-UniformConstant-04655**  
  Any variable in the *UniformConstant* storage class must be typed as either *OpTypeImage, OpTypeSampler, OpTypeSampledImage, OpTypeAccelerationStructureKHR*, or an array of one of these types.

• **VUID-StandaloneSpirv-OpTypeImage-04656**  
  *OpTypeImage* must declare a scalar 32-bit float, 64-bit integer, or 32-bit integer type for the “Sampled Type” (*RelaxedPrecision* can be applied to a sampling instruction and to the variable holding the result of a sampling instruction).

• **VUID-StandaloneSpirv-OpTypeImage-04657**  
  *OpTypeImage* must have a “Sampled” operand of 1 (sampled image) or 2 (storage image).

• **VUID-StandaloneSpirv-OpImageTexturePointer-04658**  
  If an *OpImageTexturePointer* is used in an atomic operation, the image type of the *image* parameter to *OpImageTexturePointer* must have an image format of *R64i, R64ui, R32f, R32i*, or *R32ui*.

• **VUID-StandaloneSpirv-OpImageQuerySizeLod-04659**  
  *OpImageQuerySizeLod, OpImageQueryLod, and OpImageQueryLevels* must only consume an “Image” operand whose type has its “Sampled” operand set to 1.
The (u,v) coordinates used for a SubpassData must be the <id> of a constant vector (0,0), or if a layer coordinate is used, must be a vector that was formed with constant 0 for the u and v components.

Objects of types OpTypeImage, OpTypeSampler, OpTypeSampledImage, and arrays of these types must not be stored to or modified.

Any image operation must use at most one of the Offset, ConstOffset, and ConstOffsets image operands.

Image operand Offset must only be used with OpImage*Gather instructions.

Any image instruction which uses an Offset, ConstOffset, or ConstOffsets image operand, must only consume a “Sampled Image” operand whose type has its “Sampled” operand set to 1.

The “Component” operand of OpImageGather, and OpImageSparseGather must be the <id> of a constant instruction.

OpImage*Dref must not consume an image whose Dim is 3D.

Objects of types OpTypeAccelerationStructureKHR and arrays of this type must not be stored to or modified.

The value of the “Hit Kind” operand of OpReportIntersectionKHR must be in the range [0,127].

Structure types must not contain opaque types.

Any BuiltIn decoration not listed in Built-In Variables must not be used.

The GLSLSShared and GLSLPacked decorations must not be used.

The Flat, NoPerspective, Sample, and Centroid decorations must not be used on variables with storage class other than Input or on variables used in the interface of non-fragment shader entry points.

The Flat decorations must be used on variables with storage class of Input in a fragment shader stage that are a scalar integer, vector of integer, or any double-precision floating-point type.
The **Patch** decoration **must** not be used on variables in the interface of a vertex, geometry, or fragment shader stage's entry point

- **VUID-StandaloneSpirv-ViewportRelativeNV-04672**
  The **ViewportRelativeNV** decoration **must** only be used on a variable decorated with **Layer** in the vertex, tessellation evaluation, or geometry shader stages

- **VUID-StandaloneSpirv-ViewportRelativeNV-04673**
  The **ViewportRelativeNV** decoration **must** not be used unless a variable decorated with one of **ViewportIndex** or **ViewportMaskNV** is also statically used by the same **OpEntryPoint**

- **VUID-StandaloneSpirv-ViewportMaskNV-04674**
  The **ViewportMaskNV** and **ViewportIndex** decorations **must** not both be statically used by one or more **OpEntryPoint**'s that form the vertex processing stages of a graphics pipeline

- **VUID-StandaloneSpirv-FPRoundingMode-04675**
  Rounding modes other than round-to-nearest-even and round-towards-zero **must** not be used for the **FPRoundingMode** decoration

- **VUID-StandaloneSpirv-FPRoundingMode-04676**
  The **FPRoundingMode** decoration **must** only be used for a width-only conversion instruction whose only uses are **Object** operands of **OpStore** instructions storing through a pointer to a 16-bit floating-point object in the **StorageBuffer**, **PhysicalStorageBuffer**, **Uniform**, or **Output** storage class

- **VUID-StandaloneSpirv-Invariant-04677**
  Variables decorated with **Invariant** and variables with structure types that have any members decorated with **Invariant** **must** be in the **Output** or **Input** storage class, **Invariant** used on an **Input** storage class variable or structure member has no effect

- **VUID-StandaloneSpirv-VulkanMemoryModel-04678**
  If the **VulkanMemoryModel** capability is not declared, the **Volatile** decoration **must** be used on any variable declaration that includes one of the **SMIDNV**, **WarpIDNV**, **SubgroupSize**, **SubgroupLocalInvocationId**, **SubgroupEqMask**, **SubgroupGeMask**, **SubgroupGtMask**, **SubgroupLeMask**, or **SubgroupLtMask BuiltIn** decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the **RayTmaxKHR Builtin** decoration when used in an intersection shader

- **VUID-StandaloneSpirv-VulkanMemoryModel-04679**
  If the **VulkanMemoryModel** capability is declared, the **OpLoad** instruction **must** use the **Volatile** memory semantics when it accesses into any variable that includes one of the **SMIDNV**, **WarpIDNV**, **SubgroupSize**, **SubgroupLocalInvocationId**, **SubgroupEqMask**, **SubgroupGeMask**, **SubgroupGtMask**, **SubgroupLeMask**, or **SubgroupLtMask BuiltIn** decorations when used in the ray generation, closest hit, miss, intersection, or callable shaders, or with the **RayTmaxKHR Builtin** decoration when used in an intersection shader

- **VUID-StandaloneSpirv-OpTypeRuntimeArray-04680**
  **OpTypeRuntimeArray** **must** only be used for the last member of an **OpTypeStruct** that is in the **StorageBuffer** or **PhysicalStorageBuffer** storage class decorated as **Block**, or that is in the **Uniform** storage class decorated as **BufferBlock**

- **VUID-StandaloneSpirv-Function-04681**
  A type **T** that is an array sized with a specialization constant **must** neither be, nor be contained in, the type **T2** of a variable **V**, unless either: a) **T** is equal to **T2**, b) **V** is declared
in the **Function**, or **Private** storage classes, c) \(V\) is a non-Block variable in the **Workgroup** storage class, or d) \(V\) is an interface variable with an additional level of arrayness, as described in interface matching, and \(T\) is the member type of the array type \(T2\).

- **VUID-StandaloneSpirv-OpControlBarrier-04682**
  
  If **OpControlBarrier** is used in ray generation, intersection, any-hit, closest hit, miss, fragment, vertex, tessellation evaluation, or geometry shaders, the execution Scope must be **Subgroup**

- **VUID-StandaloneSpirv-LocalSize-04683**
  
  For each compute shader entry point, either a **LocalSize** execution mode or an object decorated with the **WorkgroupSize** decoration must be specified

- **VUID-StandaloneSpirv-DerivativeGroupQuadsNV-04684**
  
  For compute shaders using the **DerivativeGroupQuadsNV** execution mode, the first two dimensions of the local workgroup size must be a multiple of two

- **VUID-StandaloneSpirv-DerivativeGroupLinearNV-04778**
  
  For compute shaders using the **DerivativeGroupLinearNV** execution mode, the product of the dimensions of the local workgroup size must be a multiple of four

- **VUID-StandaloneSpirv-OpGroupNonUniformBallotBitCount-04685**
  
  If **OpGroupNonUniformBallotBitCount** is used, the group operation must be limited to **Reduce**, **InclusiveScan**, or **ExclusiveScan**

- **VUID-StandaloneSpirv-None-04686**
  
  The **Pointer** operand of all atomic instructions must have a **Storage Class** limited to **Uniform**, **Workgroup**, **Image**, **StorageBuffer**, or **PhysicalStorageBuffer**

- **VUID-StandaloneSpirv-Offset-04687**
  
  Output variables or block members decorated with **Offset** that have a 64-bit type, or a composite type containing a 64-bit type, must specify an **Offset** value aligned to a 8 byte boundary

- **VUID-StandaloneSpirv-Offset-04689**
  
  The size of any output block containing any member decorated with **Offset** that is a 64-bit type must be a multiple of 8

- **VUID-StandaloneSpirv-Offset-04690**
  
  The first member of an output block that specifies a **Offset** decoration must specify a **Offset** value that is aligned to an 8 byte boundary if that block contains any member decorated with **Offset** and is a 64-bit type

- **VUID-StandaloneSpirv-Offset-04691**
  
  Output variables or block members decorated with **Offset** that have a 32-bit type, or a composite type contains a 32-bit type, must specify an **Offset** value aligned to a 4 byte boundary

- **VUID-StandaloneSpirv-Offset-04692**
  
  Output variables, blocks or block members decorated with **Offset** must only contain base types that have components that are either 32-bit or 64-bit in size

- **VUID-StandaloneSpirv-Offset-04716**
  
  Only variables or block members in the output interface decorated with **Offset** can be captured for transform feedback, and those variables or block members must also be...
decorated with \textit{XfbBuffer} and \textit{XfbStride}, or inherit \textit{XfbBuffer} and \textit{XfbStride} decorations from a block containing them

- \textbf{VUID-StandaloneSpirv-XfbBuffer-04693}
  All variables or block members in the output interface of the entry point being compiled decorated with a specific \textit{XfbBuffer} value \textbf{must} all be decorated with identical \textit{XfbStride} values

- \textbf{VUID-StandaloneSpirv-Stream-04694}
  If any variables or block members in the output interface of the entry point being compiled are decorated with \textit{Stream}, then all variables belonging to the same \textit{XfbBuffer} \textbf{must} specify the same \textit{Stream} value

- \textbf{VUID-StandaloneSpirv-XfbBuffer-04696}
  For any two variables or block members in the output interface of the entry point being compiled with the same \textit{XfbBuffer} value, the ranges determined by the \textit{Offset} decoration and the size of the type \textbf{must} not overlap

- \textbf{VUID-StandaloneSpirv-RayPayloadKHR-04698}
  \textit{RayPayloadKHR} storage class \textbf{must} only be used in ray generation, any-hit, closest hit or miss shaders

- \textbf{VUID-StandaloneSpirv-IncomingRayPayloadKHR-04699}
  \textit{IncomingRayPayloadKHR} storage class \textbf{must} only be used in closest hit, any-hit, or miss shaders

- \textbf{VUID-StandaloneSpirv-IncomingRayPayloadKHR-04700}
  There \textbf{must} be at most one variable with the \textit{IncomingRayPayloadKHR} storage class in the input interface of an entry point

- \textbf{VUID-StandaloneSpirv-HitAttributeKHR-04701}
  \textit{HitAttributeKHR} storage class \textbf{must} only be used in intersection, any-hit, or closest hit shaders

- \textbf{VUID-StandaloneSpirv-HitAttributeKHR-04702}
  There \textbf{must} be at most one variable with the \textit{HitAttributeKHR} storage class in the input interface of an entry point

- \textbf{VUID-StandaloneSpirv-HitAttributeKHR-04703}
  A variable with \textit{HitAttributeKHR} storage class \textbf{must} only be written to in an intersection shader

- \textbf{VUID-StandaloneSpirv-CallableDataKHR-04704}
  \textit{CallableDataKHR} storage class \textbf{must} only be used in ray generation, closest hit, miss, and callable shaders

- \textbf{VUID-StandaloneSpirv-IncomingCallableDataKHR-04705}
  \textit{IncomingCallableDataKHR} storage class \textbf{must} only be used in callable shaders

- \textbf{VUID-StandaloneSpirv-IncomingCallableDataKHR-04706}
  There \textbf{must} be at most one variable with the \textit{IncomingCallableDataKHR} storage class in the
input interface of an entry point

- VUID-StandaloneSpirv-Base-04707
  The **Base** operand of **OpPtrAccessChain** must point to one of the following: **Workgroup**, if **VariablePointers** is enabled; **StorageBuffer**, if **VariablePointers** or **VariablePointersStorageBuffer** is enabled; **PhysicalStorageBuffer**, if the **PhysicalStorageBuffer64** addressing model is enabled.

- VUID-StandaloneSpirv-PhysicalStorageBuffer64-04708
  If the **PhysicalStorageBuffer64** addressing model is enabled, all instructions that support memory access operands and that use a physical pointer must include the **Aligned** operand.

- VUID-StandaloneSpirv-PhysicalStorageBuffer64-04709
  If the **PhysicalStorageBuffer64** addressing model is enabled, any access chain instruction that accesses into a **RowMajor** matrix must only be used as the **Pointer** operand to **OpLoad** or **OpStore**.

- VUID-StandaloneSpirv-PhysicalStorageBuffer64-04710
  If the **PhysicalStorageBuffer64** addressing model is enabled, **OpConvertUToPtr** and **OpConvertPtrToU** must use an integer type whose **Width** is 64.

- VUID-StandaloneSpirv-OpTypeForwardPointer-04711
  **OpTypeForwardPointer** must have a storage class of **PhysicalStorageBuffer**.

- VUID-StandaloneSpirv-None-04745
  All variables with a storage class of **PushConstant** declared as an array must only be accessed by dynamically uniform indices.

- VUID-StandaloneSpirv-Result-04780
  The **Result Type** operand of any **OpImageRead** or **OpImageSparseRead** instruction must be a vector of four components.

- VUID-StandaloneSpirv-Base-04781

### Runtime SPIR-V Validation

The following rules must be validated at runtime. These rules depend on knowledge of the implementation and its capabilities and knowledge of runtime information, such as enabled features.

- If **vulkanMemoryModel** is enabled and **vulkanMemoryModelDeviceScope** is not enabled, **Device** memory scope must not be used.
- If **vulkanMemoryModel** is not enabled, **QueueFamily** memory scope must not be used.
- If **shaderSubgroupClock** is not enabled, the **Subgroup** scope must not be used for **OpReadClockKHR**.
- If **shaderDeviceClock** is not enabled, the **Device** scope must not be used for **OpReadClockKHR**.
- The converted bit width, signedness, and numeric type of the **Image Format** operand of an **OpTypeImage** must match the **Sampled Type**, as defined in **Image Format and Type Matching**.

---

2444 | Appendix A: Vulkan Environment for SPIR-V
• If `shaderStorageImageWriteWithoutFormat` is not enabled and an `OpTypeImage` has “Image Format” operand of `Unknown`, any variables created with the given type must be decorated with `NonWritable`.

• If `shaderStorageImageReadWithoutFormat` is not enabled and an `OpTypeImage` has “Image Format” operand of `Unknown`, any variables created with the given type must be decorated with `NonReadable`.

• Any `BuiltIn` decoration that corresponds only to Vulkan features or extensions that have not been enabled must not be used.

• `OpTypeRuntimeArray` must only be used for an array of variables with storage class `Uniform`, `StorageBuffer`, or `UniformConstant`, or for the outermost dimension of an array of arrays of such variables if the `runtimeDescriptorArray` feature is enabled.

• If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not dynamically uniform, then the operand corresponding to that resource (e.g. the pointer or sampled image operand) must be decorated with `NonUniform`.

• Result Type for Non Uniform Group Operations must be limited to 32-bit floating-point, 32-bit integer, boolean, or vectors of these types.
  ◦ If the `Float64` capability is enabled, 64-bit floating-point and vector of 64-bit floating-point types are also permitted.
  ◦ If the `Int8` capability is enabled and the `shaderSubgroupExtendedTypes` feature is `VK_TRUE`, 8-bit integer and vector of 8-bit integer types are also permitted.
  ◦ If the `Int16` capability is enabled and the `shaderSubgroupExtendedTypes` feature is `VK_TRUE`, 16-bit integer and vector of 16-bit integer types are also permitted.
  ◦ If the `Int64` capability is enabled and the `shaderSubgroupExtendedTypes` feature is `VK_TRUE`, 64-bit integer and vector of 64-bit integer types are also permitted.
  ◦ If the `Float16` capability is enabled and the `shaderSubgroupExtendedTypes` feature is `VK_TRUE`, 16-bit floating-point and vector of 16-bit floating-point types are also permitted.

• `shaderBufferInt64Atomics` must be enabled for 64-bit integer atomic operations to be supported on a Pointer with a Storage Class of `StorageBuffer` or `Uniform`.

• `shaderSharedInt64Atomics` must be enabled for 64-bit integer atomic operations to be supported on a Pointer with a Storage Class of `Workgroup`.

• `shaderBufferFloat32Atomics` or `shaderBufferFloat32AtomicAdd` or `shaderBufferFloat64Atomics` or `shaderBufferFloat64AtomicAdd` must be enabled for floating-point atomic operations to be supported on a Pointer with a Storage Class of `StorageBuffer`.

• `shaderSharedFloat32Atomics` or `shaderSharedFloat32AtomicAdd` or `shaderSharedFloat64Atomics` or `shaderSharedFloat64AtomicAdd` must be enabled for floating-point atomic operations to be supported on a Pointer with a Storage Class of `Workgroup`.

• `shaderImageFloat32Atomics` or `shaderImageFloat32AtomicAdd` must be enabled for 32-bit floating-point atomic operations to be supported on a Pointer with a Storage Class of `Image`.

• `sparseImageFloat32Atomics` or `sparseImageFloat32AtomicAdd` must be enabled for 32-bit floating-point atomic operations to be supported on sparse images.
• **shaderImageInt64Atomics** must be enabled for 64-bit integer atomic operations to be supported on a **Pointer** with a **Storage Class** of **Image**.

• If `denormBehaviorIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY`, then the entry point must use the same denormals execution mode for both 16-bit and 64-bit floating-point types.

• If `denormBehaviorIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE`, then the entry point must use the same denormals execution mode for all floating-point types.

• If `roundingModeIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY`, then the entry point must use the same rounding execution mode for both 16-bit and 64-bit floating-point types.

• If `roundingModeIndependence` is `VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE`, then the entry point must use the same rounding execution mode for all floating-point types.

• If `shaderSignedZeroInfNanPreserveFloat16` is `VK_FALSE`, then **SignedZeroInfNanPreserve** for 16-bit floating-point type must not be used.

• If `shaderSignedZeroInfNanPreserveFloat32` is `VK_FALSE`, then **SignedZeroInfNanPreserve** for 32-bit floating-point type must not be used.

• If `shaderSignedZeroInfNanPreserveFloat64` is `VK_FALSE`, then **SignedZeroInfNanPreserve** for 64-bit floating-point type must not be used.

• If `shaderDenormPreserveFloat16` is `VK_FALSE`, then **DenormPreserve** for 16-bit floating-point type must not be used.

• If `shaderDenormPreserveFloat32` is `VK_FALSE`, then **DenormPreserve** for 32-bit floating-point type must not be used.

• If `shaderDenormPreserveFloat64` is `VK_FALSE`, then **DenormPreserve** for 64-bit floating-point type must not be used.

• If `shaderDenormFlushToZeroFloat16` is `VK_FALSE`, then **DenormFlushToZero** for 16-bit floating-point type must not be used.

• If `shaderDenormFlushToZeroFloat32` is `VK_FALSE`, then **DenormFlushToZero** for 32-bit floating-point type must not be used.

• If `shaderDenormFlushToZeroFloat64` is `VK_FALSE`, then **DenormFlushToZero** for 64-bit floating-point type must not be used.

• If `shaderRoundingModeRTEFloat16` is `VK_FALSE`, then **RoundingModeRTE** for 16-bit floating-point type must not be used.

• If `shaderRoundingModeRTEFloat32` is `VK_FALSE`, then **RoundingModeRTE** for 32-bit floating-point type must not be used.

• If `shaderRoundingModeRTEFloat64` is `VK_FALSE`, then **RoundingModeRTE** for 64-bit floating-point type must not be used.

• If `shaderRoundingModeRTZFloat16` is `VK_FALSE`, then **RoundingModeRTZ** for 16-bit floating-point type must not be used.

• If `shaderRoundingModeRTZFloat32` is `VK_FALSE`, then **RoundingModeRTZ** for 32-bit floating-point type must not be used.

• If `shaderRoundingModeRTZFloat64` is `VK_FALSE`, then **RoundingModeRTZ** for 64-bit floating-point type must not be used.
• If shaderRoundingModeRTZFloat64 is VK_FALSE, then RoundingModeRTZ for 64-bit floating-point type must not be used.

• The Offset plus size of the type of each variable, in the output interface of the entry point being compiled, decorated with XfbBuffer must not be greater than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferDataSize.

• For any given XfbBuffer value, define the buffer data size to be the smallest number of bytes such that, for all outputs decorated with the same XfbBuffer value, the size of the output interface variable plus the Offset is less than or equal to the buffer data size. For a given Stream, the sum of all the buffer data sizes for all buffers writing to that stream the must not exceed VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreamDataSize.

• The Stream value to OpEmitStreamVertex and OpEndStreamPrimitive must be less than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams.

• If the geometry shader emits to more than one vertex stream and VkPhysicalDeviceTransformFeedbackPropertiesEXT::transformFeedbackStreamsLinesTriangles is VK_FALSE, then execution mode must be OutputPoints.

• The stream number value to Stream must be less than VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackStreams.

• The XFB Stride value to XfbStride must be less than or equal to VkPhysicalDeviceTransformFeedbackPropertiesEXT::maxTransformFeedbackBufferDataStride.

• If the PhysicalStorageBuffer64 addressing model is enabled any load or store through a physical pointer type must be aligned to a multiple of the size of the largest scalar type in the pointed-to type.

• If the PhysicalStorageBuffer64 addressing model is enabled the pointer value of a memory access instruction must be at least as aligned as specified by the Aligned memory access operand.

• For OpTypeCooperativeMatrixNV, the component type, scope, number of rows, and number of columns must match one of the matrices in any of the supported VkCooperativeMatrixPropertiesNV.

• For OpCooperativeMatrixMulAddNV, the type of A must have VkCooperativeMatrixPropertiesNV::MSize rows and VkCooperativeMatrixPropertiesNV::KSize columns and have a component type that matches VkCooperativeMatrixPropertiesNV::AType.

• For OpCooperativeMatrixMulAddNV, the type of B must have VkCooperativeMatrixPropertiesNV::KSize rows and VkCooperativeMatrixPropertiesNV::NSize columns and have a component type that matches VkCooperativeMatrixPropertiesNV::BType.

• For OpCooperativeMatrixMulAddNV, the type of C must have VkCooperativeMatrixPropertiesNV::MSize rows and VkCooperativeMatrixPropertiesNV::NSize columns and have a component type that matches VkCooperativeMatrixPropertiesNV::CType.

• For OpCooperativeMatrixMulAddNV, the type of Result must have VkCooperativeMatrixPropertiesNV::MSize rows and VkCooperativeMatrixPropertiesNV::NSize columns and have a component type that matches VkCooperativeMatrixPropertiesNV::DType.

• For OpCooperativeMatrixMulAddNV, the type of A, B, C, and Result must all have a scope of scope.

• OpTypeCooperativeMatrixNV and OpCooperativeMatrix* instructions must not be used in shader.
stages not included in VkPhysicalDeviceCooperativeMatrixPropertiesNV::cooperativeMatrixSupportedStages.

- DescriptorSet and Binding decorations must obey the constraints on storage class, type, and descriptor type described in DescriptorSet and Binding Assignment.

- For OpCooperativeMatrixLoadNV and OpCooperativeMatrixStoreNV instructions, the Pointer and Stride operands must be aligned to at least the lesser of 16 bytes or the natural alignment of a row or column (depending on ColumnMajor) of the matrix (where the natural alignment is the number of columns/rows multiplied by the component size).

- If the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::shaderSampleRateInterpolationFunctions is VK_FALSE, then GLSL.std.450 fragment interpolation functions are not supported by the implementation and OpCapability must not be set to InterpolationFunction.

- If tessellationShader is enabled, and the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::tessellationIsolines is VK_FALSE, then OpExecutionMode must not be set to Isolines.

- If tessellationShader is enabled, and the VK_KHR_portability_subset extension is enabled, and VkPhysicalDevicePortabilitySubsetFeaturesKHR::tessellationPointMode is VK_FALSE, then OpExecutionMode must not be set to PointMode.

- If storageBuffer8BitAccess is VK_FALSE, then objects containing an 8-bit integer element must not have storage class of StorageBuffer, ShaderRecordBufferKHR, or PhysicalStorageBuffer.

- If uniformAndStorageBuffer8BitAccess is VK_FALSE, then objects in the Uniform storage class with the Block decoration must not have an 8-bit integer member.

- If storagePushConstant8 is VK_FALSE, then objects containing an 8-bit integer element must not have storage class of PushConstant.

- If storageBuffer16BitAccess is VK_FALSE, then objects containing 16-bit integer or 16-bit floating-point elements must not have storage class of StorageBuffer, ShaderRecordBufferKHR, or PhysicalStorageBuffer.

- If uniformAndStorageBuffer16BitAccess is VK_FALSE, then objects in the Uniform storage class with the Block decoration must not have 16-bit integer or 16-bit floating-point members.

- If storagePushConstant16 is VK_FALSE, then objects containing 16-bit integer or 16-bit floating-point elements must not have storage class of PushConstant.

- If storageInputOutput16 is VK_FALSE, then objects containing 16-bit integer or 16-bit floating-point elements must not have storage class of Input or Output.

- Atomic instructions must declare a scalar 32-bit integer type, or a scalar 32-bit floating-point type if the shaderBufferFloat32Atomics or shaderBufferFloat32AtomicAdd or shaderSharedFloat32Atomics or shaderSharedFloat32AtomicAdd or shaderImageFloat32Atomics or shaderImageFloat32AtomicAdd or sparseImageFloat32AtomicAdd is enabled, or a scalar 64-bit floating-point type if the shaderBufferFloat64Atomics or shaderBufferFloat64AtomicAdd or shaderSharedFloat64Atomics or shaderSharedFloat64AtomicAdd is enabled, or a scalar 64-bit integer type if the Int64Atomics capability is enabled, for the value pointed to by Pointer.

- If fragmentStoresAndAtomics is not enabled, then all storage image, storage texel buffer, and...
storage buffer variables in the fragment stage must be decorated with the NonWritable decoration.

- If vertexPipelineStoresAndAtomics is not enabled, then all storage image, storage texel buffer, and storage buffer variables in the vertex, tessellation, and geometry stages must be decorated with the NonWritable decoration.

- If subgroupQuadOperationsInAllStages is VK_FALSE, then quad subgroup operations must not be used except for in fragment and compute stages.

- Group operations with subgroup scope must not be used if the shader stage is not in subgroupSupportedStages.

- The first element of the Offset operand of InterpolateAtOffset must be greater than or equal to:

\[
\text{frag}_{\text{width}} \times \text{minInterpolationOffset}
\]

where \(\text{frag}_{\text{width}}\) is the width of the current fragment in pixels.

- The first element of the Offset operand of InterpolateAtOffset must be less than or equal to:

\[
\text{frag}_{\text{width}} \times (\text{maxInterpolationOffset} + \text{ULP}) - \text{ULP}
\]

where \(\text{frag}_{\text{width}}\) is the width of the current fragment in pixels and \(\text{ULP} = 1 / 2^{\text{subPixelInterpolationOffsetBits}}\).

- The second element of the Offset operand of InterpolateAtOffset must be greater than or equal to:

\[
\text{frag}_{\text{height}} \times \text{minInterpolationOffset}
\]

where \(\text{frag}_{\text{height}}\) is the height of the current fragment in pixels.

- The second element of the Offset operand of InterpolateAtOffset must be less than or equal to:

\[
\text{frag}_{\text{height}} \times (\text{maxInterpolationOffset} + \text{ULP}) - \text{ULP}
\]

where \(\text{frag}_{\text{height}}\) is the height of the current fragment in pixels and \(\text{ULP} = 1 / 2^{\text{subPixelInterpolationOffsetBits}}\).

- For OpRayQueryInitializeKHR instructions, all components of the RayOrigin and RayDirection operands must be finite floating-point values.

- For OpRayQueryInitializeKHR instructions, the RayTmin and RayTmax operands must be non-negative floating-point values.

- For OpRayQueryInitializeKHR instructions, the RayTmin operand must be less than or equal to the RayTmax operand.

- For OpRayQueryInitializeKHR instructions, RayOrigin, RayDirection, RayTmin, and RayTmax operands
must not contain NaNs.

- For OpRayQueryInitializeKHR instructions, Acceleration Structure must be an acceleration structure built as a top-level acceleration structure.

- For OpRayQueryGenerateIntersectionKHR instructions, Hit T must satisfy the condition RayTmin ≤ Hit T ≤ RayTmax, where RayTmin is equal to the value returned by OpRayQueryGetRayTMinKHR with the same ray query object, and RayTmax is equal to the value of OpRayQueryGetIntersectionTKHR for the current committed intersection with the same ray query object.

- For OpTraceRayKHR instructions, all components of the RayOrigin and RayDirection operands must be finite floating-point values.

- For OpTraceRayKHR instructions, the RayTmin and RayTmax operands must be non-negative floating-point values.

- For OpTraceRayKHR instructions, the RayTmin operand must be less than or equal to the RayTmax operand.

- For OpTraceRayKHR instructions, RayOrigin, RayDirection, RayTmin, and RayTmax operands must not contain NaNs.

- For OpTraceRayKHR instructions, Acceleration Structure must be an acceleration structure built as a top-level acceleration structure.

  - The x size in LocalSize must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupSize[0]
  
  - The y size in LocalSize must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupSize[1]
  
  - The z size in LocalSize must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupSize[2]

  - The product of x size, y size, and z size in LocalSize must be less than or equal to VkPhysicalDeviceLimits::maxComputeWorkGroupInvocations

- If shaderZeroInitializeWorkgroupMemory is not enabled, any OpVariable with Workgroup as its Storage Class must not have an Initializer operand

- If workgroupMemoryExplicitLayout8BitAccess is VK_FALSE, objects in the Workgroup storage class with the Block decoration must not contain 8-bit integer members.

- If workgroupMemoryExplicitLayout16BitAccess is VK_FALSE, objects in the Workgroup storage class with the Block decoration must not contain 16-bit integer or 16-bit floating-point members.

- If an OpImage*Gather operation has an image operand of Offset, ConstOffset, or ConstOffsets the offset value must be greater than or equal to minTexelGatherOffset

- If an OpImage*Gather operation has an image operand of Offset, ConstOffset, or ConstOffsets the offset value must be less than or equal to maxTexelGatherOffset

**Precision and Operation of SPIR-V Instructions**

The following rules apply to half, single, and double-precision floating point instructions:

- Positive and negative infinities and positive and negative zeros are generated as dictated by
IEEE 754, but subject to the precisions allowed in the following table.

- Dividing a non-zero by a zero results in the appropriately signed IEEE 754 infinity.
- Signaling NaNs are not required to be generated and exceptions are never raised. Signaling NaN may be converted to quiet NaNs values by any floating point instruction.
- By default, the implementation may perform optimizations on half, single, or double-precision floating-point instructions that ignore sign of a zero, or assume that arguments and results are not NaNs or infinities. If the entry point is declared with the SignedZeroInfNanPreserve execution mode, then NaNs, infinities, and the sign of zero must not be ignored.


- Denormalized values are supported.
  - By default, any half, single, or double-precision denormalized value input into a shader or potentially generated by any instruction (except those listed above) or any extended instructions for GLSL in a shader may be flushed to zero.
  - If the entry point is declared with the DenormFlushToZero execution mode then for the affected instructions the denormalized result must be flushed to zero and the denormalized operands may be flushed to zero. Denormalized values obtained via unpacking an integer into a vector of values with smaller bit width and interpreting those values as floating-point numbers must be flushed to zero.


The precision of double-precision instructions is at least that of single precision.

The precision of operations is defined either in terms of rounding, as an error bound in ULP, or as inherited from a formula as follows.

**Correctly Rounded**

Operations described as “correctly rounded” will return the infinitely precise result, \( x \), rounded so as to be representable in floating-point. The rounding mode is not specified, unless the entry point is declared with the `RoundingModeRTE` or the `RoundingModeRTZ` execution mode. These execution modes affect only correctly rounded SPIR-V instructions. These execution modes do not affect `OpQuantizeToF16`. If the rounding mode is not specified then this rounding is implementation specific, subject to the following rules. If \( x \) is exactly representable then \( x \) will be returned. Otherwise, either the floating-point value closest to and no less than \( x \) or the value closest to and no greater than \( x \) will be returned.

**ULP**

Where an error bound of \( n \) ULP (units in the last place) is given, for an operation with infinitely precise result \( x \) the value returned must be in the range \([x - n \times \text{ulp}(x), x + n \times \text{ulp}(x)]\). The function \( \text{ulp}(x) \) is defined as follows:

If there exist non-equal floating-point numbers \( a \) and \( b \) such that \( a \leq x \leq b \) then \( \text{ulp}(x) \) is the minimum possible distance between such numbers, \( \text{ulp}(x) = \min_a, |b - a| \). If such numbers do not exist then \( \text{ulp}(x) \) is defined to be the difference between the two finite floating-point numbers nearest to \( x \).

Where the range of allowed return values includes any value of magnitude larger than that of the largest representable finite floating-point number, operations may, additionally, return either an infinity of the appropriate sign or the finite number with the largest magnitude of the appropriate sign. If the infinitely precise result of the operation is not mathematically defined then the value returned is undefined.

**Inherited From ...**

Where an operation’s precision is described as being inherited from a formula, the result returned must be at least as accurate as the result of computing an approximation to \( x \) using a formula equivalent to the given formula applied to the supplied inputs. Specifically, the formula given may be transformed using the mathematical associativity, commutativity and distributivity of the operators involved to yield an equivalent formula. The SPIR-V precision rules, when applied to each such formula and the given input values, define a range of permitted values. If NaN is one of the permitted values then the operation may return any result, otherwise let the largest permitted value in any of the ranges be \( F_{\text{max}} \) and the smallest be \( F_{\text{min}} \). The operation must return a value in the range \([x - E, x + E]\) where \( E = \max(\|x - F_{\text{min}}\|, \|x - F_{\text{max}}\|) \). If the entry point is declared with the
DenormFlushToZero execution mode, then any intermediate denormal value(s) while evaluating the formula may be flushed to zero. Denormal final results must be flushed to zero. If the entry point is declared with the DenormPreserve execution mode, then denormals must be preserved throughout the formula.

For half- (16 bit) and single- (32 bit) precision instructions, precisions are required to be at least as follows:

### Table 85. Precision of core SPIR-V Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single precision, unless decorated with RelaxedPrecision</th>
<th>Half precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpFAdd</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>OpFSub</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>OpDot(x, y)</td>
<td>Inherited from $\sum_{i=0}^{n-1} x_i \times y_i$.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdEqual, OpFUnordEqual</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdLessThan, OpFUnordLessThan</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdGreaterThan, OpFUnordGreaterThan</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFOrdGreaterThanEqual, OpFUnordGreaterThanEqual</td>
<td>Correct result.</td>
<td></td>
</tr>
<tr>
<td>OpFDiv(x,y)</td>
<td>2.5 ULP for $</td>
<td>y</td>
</tr>
<tr>
<td>OpFRem(x,y)</td>
<td>Inherited from $x - y \times \text{trunc}(x/y)$.</td>
<td></td>
</tr>
<tr>
<td>OpFMod(x,y)</td>
<td>Inherited from $x - y \times \text{floor}(x/y)$.</td>
<td></td>
</tr>
<tr>
<td>conversions between types</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

The OpFRem and OpFMod instructions use cheap approximations of remainder, and the error can be large due to the discontinuity in \text{trunc()} and \text{floor()}. This can produce mathematically unexpected results in some cases, such as FMod(x,x) computing x rather than 0, and can also cause the result to have a different sign than the infinitely precise result.

### Table 86. Precision of GLSL.std.450 Instructions
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Single precision, unless decorated with RelaxedPrecision</th>
<th>Half precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>fma()</td>
<td>Inherited from OpFMul followed by OpFAdd.</td>
<td></td>
</tr>
<tr>
<td>exp(x), exp2(x)</td>
<td>$3 + 2 \times</td>
<td>x</td>
</tr>
<tr>
<td>log(), log2()</td>
<td>3 ULP outside the range $[0.5, 2.0]$. Absolute error $&lt; 2^{-21}$ inside the range $[0.5, 2.0]$.</td>
<td>3 ULP outside the range $[0.5, 2.0]$. Absolute error $&lt; 2^{-7}$ inside the range $[0.5, 2.0]$.</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>Inherited from exp$2(y \times \log2(x))$.</td>
<td></td>
</tr>
<tr>
<td>sqrt()</td>
<td>Inherited from 1.0 / inversesqrt().</td>
<td></td>
</tr>
<tr>
<td>inversesqrt()</td>
<td>2 ULP.</td>
<td></td>
</tr>
<tr>
<td>radians(x)</td>
<td>Inherited from $\times \frac{\pi}{180}$.</td>
<td></td>
</tr>
<tr>
<td>degrees(x)</td>
<td>Inherited from $\times \frac{180}{\pi}$.</td>
<td></td>
</tr>
<tr>
<td>sin()</td>
<td>Absolute error $\leq 2^{-11}$ inside the range $[-n, n]$.</td>
<td>Absolute error $\leq 2^{-7}$ inside the range $[-n, n]$.</td>
</tr>
<tr>
<td>cos()</td>
<td>Absolute error $\leq 2^{-11}$ inside the range $[-n, n]$.</td>
<td>Absolute error $\leq 2^{-7}$ inside the range $[-n, n]$.</td>
</tr>
<tr>
<td>tan()</td>
<td>Inherited from $\frac{\sin()}{\cos()}$.</td>
<td></td>
</tr>
<tr>
<td>asin(x)</td>
<td>Inherited from atan$2(x, \sqrt{1.0 - x \times x})$.</td>
<td></td>
</tr>
<tr>
<td>acos(x)</td>
<td>Inherited from atan$2(\sqrt{1.0 - x \times x}, x)$.</td>
<td></td>
</tr>
<tr>
<td>atan(), atan2()</td>
<td>4096 ULP</td>
<td>5 ULP.</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>Inherited from $(\exp(x) - \exp(-x)) \times 0.5$.</td>
<td></td>
</tr>
<tr>
<td>cosh(x)</td>
<td>Inherited from $(\exp(x) + \exp(-x)) \times 0.5$.</td>
<td></td>
</tr>
<tr>
<td>tanh()</td>
<td>Inherited from $\frac{\sinh()}{\cosh()}$.</td>
<td></td>
</tr>
<tr>
<td>asinh(x)</td>
<td>Inherited from $\log(x + \sqrt{x \times x + 1.0})$.</td>
<td></td>
</tr>
<tr>
<td>acosh(x)</td>
<td>Inherited from $\log(x + \sqrt{x \times x - 1.0})$.</td>
<td></td>
</tr>
<tr>
<td>atanh(x)</td>
<td>Inherited from $\log(\frac{1.0 + x}{1.0 - x}) \times 0.5$.</td>
<td></td>
</tr>
<tr>
<td>frexp()</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Single precision, unless decorated with RelaxedPrecision</td>
<td>Half precision</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ldexp()</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>length(x)</td>
<td>Inherited from $\sqrt{\text{dot}(x, x)}$.</td>
<td></td>
</tr>
<tr>
<td>distance(x, y)</td>
<td>Inherited from $\text{length}(x - y)$.</td>
<td></td>
</tr>
<tr>
<td>cross()</td>
<td>Inherited from $\text{OpFSub(OpFMul, OpFMul)}$.</td>
<td></td>
</tr>
<tr>
<td>normalize(x)</td>
<td>Inherited from $\frac{x}{\text{length}(x)}$.</td>
<td></td>
</tr>
<tr>
<td>faceforward(N, I, NRef)</td>
<td>Inherited from $\text{dot}(\text{NRef}, I) &lt; 0.0 ? N : -N$.</td>
<td></td>
</tr>
<tr>
<td>reflect(x, y)</td>
<td>Inherited from $x - 2.0 \times \text{dot}(y, x) \times y$.</td>
<td></td>
</tr>
<tr>
<td>refract(I, N, eta)</td>
<td>Inherited from $k &lt; 0.0 ? 0.0 : \eta \times I - (\eta \times \text{dot}(N, I) + \sqrt{k}) \times N$, where $k = 1 - \eta \times \eta \times (1.0 - \text{dot}(N, I) \times \text{dot}(N, I))$.</td>
<td></td>
</tr>
<tr>
<td>round</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>roundEven</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>trunc</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fabs</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fsign</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>ceil</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fract</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>modf</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fmin</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fmax</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fclamp</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>fmix(x, y, a)</td>
<td>Inherited from $x \times (1.0 - a) + y \times a$.</td>
<td></td>
</tr>
<tr>
<td>step</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>smoothStep(edge0, edge1, x)</td>
<td>Inherited from $t \times t \times (3.0 - 2.0 \times t)$, where $t = \text{clamp}(x - \frac{\text{edge0}}{\text{edge1} - \text{edge0}}, 0.0, 1.0)$</td>
<td></td>
</tr>
<tr>
<td>nmin</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>nmax</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
<tr>
<td>nclamp</td>
<td>Correctly rounded.</td>
<td></td>
</tr>
</tbody>
</table>

GLSL.std.450 extended instructions specifically defined in terms of the above instructions inherit the above errors. GLSL.std.450 extended instructions not listed above and not defined in terms of...
the above have undefined precision.

For the `OpSRem` and `OpSMod` instructions, if either operand is negative the result is undefined.

**Note**
While the `OpSRem` and `OpSMod` instructions are supported by the Vulkan environment, they require non-negative values and thus do not enable additional functionality beyond what `OpUMod` provides.

`OpCooperativeMatrixMulAddNV` performs its operations in an implementation-dependent order and internal precision.

### Image Format and Type Matching

When specifying the Image Format as anything other than Unknown, the converted bit width, type, and signedness as shown in the table below, **must** match the Sampled Type.

**Note**
Formatted accesses are always converted from a shader readable type to the resource's format or vice versa via Format Conversion for reads and Texel Output Format Conversion for writes. As such, the bit width and format below do not necessarily match 1:1 with what might be expected for some formats.

For a given Image Format, the Sampled Type **must** be the type described in the Type column of the below table, with its Literal Width set to that in the Bit Width column, and its Literal Signedness to that in the Signedness column (where applicable).

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Type</th>
<th>Bit Width</th>
<th>Signedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Image Format</td>
<td>Type</td>
<td>Bit Width</td>
<td>Signedness</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>Rgba32f</td>
<td>OpTypeFloat</td>
<td>32</td>
<td>N/A</td>
</tr>
<tr>
<td>Rg32f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgb10A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11fG11fB10f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rgba8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rg8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8Snorm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compatibility Between SPIR-V Image Formats And Vulkan Formats

SPIR-V Image Format values are compatible with VkFormat values as defined below:

Table 87. SPIR-V and Vulkan Image Format Compatibility

<table>
<thead>
<tr>
<th>SPIR-V Image Format</th>
<th>Compatible Vulkan Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>Any</td>
</tr>
<tr>
<td>RGBA32f</td>
<td>VK_FORMAT_R32G32B32A32_SFLOAT</td>
</tr>
<tr>
<td>RGBA16f</td>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
</tr>
<tr>
<td>R32f</td>
<td>VK_FORMAT_R32_SFLOAT</td>
</tr>
<tr>
<td>RGBA8</td>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
</tr>
<tr>
<td>RGBA8SNorm</td>
<td>VK_FORMAT_R8G8B8A8_SNORM</td>
</tr>
<tr>
<td>R32f</td>
<td>VK_FORMAT_R32_SFLOAT</td>
</tr>
<tr>
<td>RGBA8</td>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
</tr>
<tr>
<td>RGBA8SNorm</td>
<td>VK_FORMAT_R8G8B8A8_SNORM</td>
</tr>
<tr>
<td>R32f</td>
<td>VK_FORMAT_R32G32:SFLOAT</td>
</tr>
<tr>
<td>R16f</td>
<td>VK_FORMAT_R16G16_SFLOAT</td>
</tr>
<tr>
<td>R16f</td>
<td>VK_FORMAT_R16G16_SFLOAT</td>
</tr>
<tr>
<td>RGBA16</td>
<td>VK_FORMAT_R16G16B16A16_UNORM</td>
</tr>
</tbody>
</table>

2458 | Appendix A: Vulkan Environment for SPIR-V
<table>
<thead>
<tr>
<th>SPIR-V Image Format</th>
<th>Compatible Vulkan Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rgb10A2</td>
<td>VK_FORMAT_A2B10G10R10_UNORM_PACK32</td>
</tr>
<tr>
<td>Rg16</td>
<td>VK_FORMAT_R16G16_UNORM</td>
</tr>
<tr>
<td>Rg8</td>
<td>VK_FORMAT_R8G8_UNORM</td>
</tr>
<tr>
<td>R16</td>
<td>VK_FORMAT_R16_UNORM</td>
</tr>
<tr>
<td>R8</td>
<td>VK_FORMAT_R8_UNORM</td>
</tr>
<tr>
<td>Rgba16Snorm</td>
<td>VK_FORMAT_R16G16B16A16_SNORM</td>
</tr>
<tr>
<td>Rg16Snorm</td>
<td>VK_FORMAT_R16G16_SNORM</td>
</tr>
<tr>
<td>Rg8Snorm</td>
<td>VK_FORMAT_R8G8_SNORM</td>
</tr>
<tr>
<td>R16Snorm</td>
<td>VK_FORMAT_R16_SNORM</td>
</tr>
<tr>
<td>R8Snorm</td>
<td>VK_FORMAT_R8_SNORM</td>
</tr>
<tr>
<td>Rgba32i</td>
<td>VK_FORMAT_R32G32B32A32_SINT</td>
</tr>
<tr>
<td>Rgba16i</td>
<td>VK_FORMAT_R16G16B16A16_SINT</td>
</tr>
<tr>
<td>Rgba8i</td>
<td>VK_FORMAT_R8G8B8A8_SINT</td>
</tr>
<tr>
<td>R32i</td>
<td>VK_FORMAT_R32_SINT</td>
</tr>
<tr>
<td>Rg32i</td>
<td>VK_FORMAT_R32G32_SINT</td>
</tr>
<tr>
<td>Rg16i</td>
<td>VK_FORMAT_R16G16_SINT</td>
</tr>
<tr>
<td>Rg8i</td>
<td>VK_FORMAT_R8G8_SINT</td>
</tr>
<tr>
<td>R16i</td>
<td>VK_FORMAT_R16_SINT</td>
</tr>
<tr>
<td>R8i</td>
<td>VK_FORMAT_R8_SINT</td>
</tr>
<tr>
<td>Rgba32ui</td>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
</tr>
<tr>
<td>Rgba16ui</td>
<td>VK_FORMAT_R16G16B16A16_UINT</td>
</tr>
<tr>
<td>Rgba8ui</td>
<td>VK_FORMAT_R8G8B8A8_UINT</td>
</tr>
<tr>
<td>R32ui</td>
<td>VK_FORMAT_R32_UINT</td>
</tr>
<tr>
<td>Rgb10a2ui</td>
<td>VK_FORMAT_A2B10G10R10_UINT_PACK32</td>
</tr>
<tr>
<td>Rg32ui</td>
<td>VK_FORMAT_R32G32_UINT</td>
</tr>
<tr>
<td>Rg16ui</td>
<td>VK_FORMAT_R16G16_UINT</td>
</tr>
<tr>
<td>Rg8ui</td>
<td>VK_FORMAT_R8G8_UINT</td>
</tr>
<tr>
<td>R16ui</td>
<td>VK_FORMAT_R16_UINT</td>
</tr>
<tr>
<td>R8ui</td>
<td>VK_FORMAT_R8_UINT</td>
</tr>
</tbody>
</table>
Appendix B: Memory Model

Agent

Operation is a general term for any task that is executed on the system.

Note
An operation is by definition something that is executed. Thus if an instruction is skipped due to control flow, it does not constitute an operation.

Each operation is executed by a particular agent. Possible agents include each shader invocation, each host thread, and each fixed-function stage of the pipeline.

Memory Location

A memory location identifies unique storage for 8 bits of data. Memory operations access a set of memory locations consisting of one or more memory locations at a time, e.g. an operation accessing a 32-bit integer in memory would read/write a set of four memory locations. Memory operations that access whole aggregates may access any padding bytes between elements or members, but no padding bytes at the end of the aggregate. Two sets of memory locations overlap if the intersection of their sets of memory locations is non-empty. A memory operation must not affect memory at a memory location not within its set of memory locations.

Memory locations for buffers and images are explicitly allocated in VkDeviceMemory objects, and are implicitly allocated for SPIR-V variables in each shader invocation.

Variables with Workgroup storage class that point to a block-decorated type share a set of memory locations.

Allocation

The values stored in newly allocated memory locations are determined by a SPIR-V variable's initializer, if present, or else are undefined. At the time an allocation is created there have been no memory operations to any of its memory locations. The initialization is not considered to be a memory operation.

Note
For tessellation control shader output variables, a consequence of initialization not being considered a memory operation is that some implementations may need to insert a barrier between the initialization of the output variables and any reads of those variables.

Memory Operation

For an operation A and memory location M:
• A reads M if and only if the data stored in M is an input to A.
• A writes M if and only if the data output from A is stored to M.
• A accesses M if and only if it either reads or writes (or both) M.

Note
A write whose value is the same as what was already in those memory locations is still considered to be a write and has all the same effects.

Reference
A reference is an object that a particular agent can use to access a set of memory locations. On the host, a reference is a host virtual address. On the device, a reference is:

• The descriptor that a variable is bound to, for variables in Image, Uniform, or StorageBuffer storage classes. If the variable is an array (or array of arrays, etc.) then each element of the array may be a unique reference.
• The address range for a buffer in PhysicalStorageBuffer storage class, where the base of the address range is queried with vkGetBufferDeviceAddress and the length of the range is the size of the buffer.
• A single common reference for all variables with Workgroup storage class that point to a block-decorated type.
• The variable itself for non-block-decorated type variables in Workgroup storage class.
• The variable itself for variables in other storage classes.

Two memory accesses through distinct references may require availability and visibility operations as defined below.

Program-Order
A dynamic instance of an instruction is defined in SPIR-V (https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.html#DynamicInstance) as a way of referring to a particular execution of a static instruction. Program-order is an ordering on dynamic instances of instructions executed by a single shader invocation:

• (Basic block): If instructions A and B are in the same basic block, and A is listed in the module before B, then the n'th dynamic instance of A is program-ordered before the n’th dynamic instance of B.
• (Branch): The dynamic instance of a branch or switch instruction is program-ordered before the dynamic instance of the OpLabel instruction to which it transfers control.
• (Call entry): The dynamic instance of an OpFunctionCall instruction is program-ordered before the dynamic instances of the OpFunctionParameter instructions and the body of the called function.
• (Call exit): The dynamic instance of the instruction following an OpFunctionCall instruction is program-ordered after the dynamic instance of the return instruction executed by the called
function.

• (Transitive Closure): If dynamic instance A of any instruction is program-ordered before dynamic instance B of any instruction and B is program-ordered before dynamic instance C of any instruction then A is program-ordered before C.

• (Complete definition): No other dynamic instances are program-ordered.

For instructions executed on the host, the source language defines the program-order relation (e.g. as “sequenced-before”).

Shader Call Related

Shader-call-related is an equivalence relation on invocations defined as the symmetric and transitive closure of:

• A is shader-call-related to B if A is created by an invocation repack instruction executed by B.

Shader Call Order

Shader-call-order is a partial order on dynamic instances of instructions executed by invocations that are shader-call-related:

• (Program order): If dynamic instance A is program-ordered before B, then A is shader-call-ordered before B.

• (Shader call entry): If A is a dynamic instance of an invocation repack instruction and B is a dynamic instance executed by an invocation that is created by A, then A is shader-call-ordered before B.

• (Shader call exit): If A is a dynamic instance of an invocation repack instruction, B is the next dynamic instance executed by the same invocation, and C is a dynamic instance executed by an invocation that is created by A, then C is shader-call-ordered before B.

• (Transitive closure): If A is shader-call-ordered-before B and B is shader-call-ordered-before C, then A is shader-call-ordered-before C.

• (Complete definition): No other dynamic instances are shader-call-ordered.

Scope

Atomic and barrier instructions include scopes which identify sets of shader invocations that must obey the requested ordering and atomicity rules of the operation, as defined below.

The various scopes are described in detail in the Shaders chapter.

Atomic Operation

An atomic operation on the device is any SPIR-V operation whose name begins with OpAtomic. An atomic operation on the host is any operation performed with an std::atomic typed object.
Each atomic operation has a memory scope and a semantics. Informally, the scope determines which other agents it is atomic with respect to, and the semantics constrains its ordering against other memory accesses. Device atomic operations have explicit scopes and semantics. Each host atomic operation implicitly uses the CrossDevice scope, and uses a memory semantics equivalent to a C++ std::memory_order value of relaxed, acquire, release, acq_rel, or seq_cst.

Two atomic operations A and B are potentially-mutually-ordered if and only if all of the following are true:

- They access the same set of memory locations.
- They use the same reference.
- A is in the instance of B’s memory scope.
- B is in the instance of A’s memory scope.
- A and B are not the same operation (irreflexive).

Two atomic operations A and B are mutually-ordered if and only if they are potentially-mutually-ordered and any of the following are true:

- A and B are both device operations.
- A and B are both host operations.
- A is a device operation, B is a host operation, and the implementation supports concurrent host-and device-atomics.

**Note**
If two atomic operations are not mutually-ordered, and if their sets of memory locations overlap, then each must be synchronized against the other as if they were non-atomic operations.

**Scoped Modification Order**
For a given atomic write A, all atomic writes that are mutually-ordered with A occur in an order known as A’s scoped modification order. A’s scoped modification order relates no other operations.

**Note**
Invocations outside the instance of A’s memory scope may observe the values at A’s set of memory locations becoming visible to it in an order that disagrees with the scoped modification order.

**Note**
It is valid to have non-atomic operations or atomics in a different scope instance to the same set of memory locations, as long as they are synchronized against each other as if they were non-atomic (if they are not, it is treated as a data race). That means this definition of A’s scoped modification order could include atomic operations that occur much later, after intervening non-atomics. That is a bit non-intuitive, but it helps to keep this definition simple and non-circular.
Memory Semantics

Non-atomic memory operations, by default, may be observed by one agent in a different order than they were written by another agent.

Atomics and some synchronization operations include memory semantics, which are flags that constrain the order in which other memory accesses (including non-atomic memory accesses and availability and visibility operations) performed by the same agent can be observed by other agents, or can observe accesses by other agents.

Device instructions that include semantics are OpAtomic*, OpControlBarrier, OpMemoryBarrier, and OpMemoryNamedBarrier. Host instructions that include semantics are some std::atomic methods and memory fences.

SPIR-V supports the following memory semantics:

- Relaxed: No constraints on order of other memory accesses.
- Acquire: A memory read with this semantic performs an acquire operation. A memory barrier with this semantic is an acquire barrier.
- Release: A memory write with this semantic performs a release operation. A memory barrier with this semantic is a release barrier.
- AcquireRelease: A memory read-modify-write operation with this semantic performs both an acquire operation and a release operation, and inherits the limitations on ordering from both of those operations. A memory barrier with this semantic is both a release and acquire barrier.

Note

SPIR-V does not support “consume” semantics on the device.

The memory semantics operand also includes storage class semantics which indicate which storage classes are constrained by the synchronization. SPIR-V storage class semantics include:

- UniformMemory
- WorkgroupMemory
- ImageMemory
- OutputMemory

Each SPIR-V memory operation accesses a single storage class. Semantics in synchronization operations can include a combination of storage classes.

The UniformMemory storage class semantic applies to accesses to memory in the PhysicalStorageBuffer, ShaderRecordBufferKHR, Uniform and StorageBuffer storage classes. The WorkgroupMemory storage class semantic applies to accesses to memory in the Workgroup storage class. The ImageMemory storage class semantic applies to accesses to memory in the Image storage class. The OutputMemory storage class semantic applies to accesses to memory in the Output storage class.
Informally, these constraints limit how memory operations can be reordered, and these limits apply not only to the order of accesses as performed in the agent that executes the instruction, but also to the order the effects of writes become visible to all other agents within the same instance of the instruction’s memory scope.

Release and acquire operations in different threads can act as synchronization operations, to guarantee that writes that happened before the release are visible after the acquire. (This is not a formal definition, just an Informative forward reference.)

The OutputMemory storage class semantic is only useful in tessellation control shaders, which is the only execution model where output variables are shared between invocations.

The memory semantics operand also optionally includes availability and visibility flags, which apply optional availability and visibility operations as described in availability and visibility. The availability/visibility flags are:

- MakeAvailable: Semantics must be Release or AcquireRelease. Performs an availability operation before the release operation or barrier.
- MakeVisible: Semantics must be Acquire or AcquireRelease. Performs a visibility operation after the acquire operation or barrier.

The specifics of these operations are defined in Availability and Visibility Semantics.

Host atomic operations may support a different list of memory semantics and synchronization operations, depending on the host architecture and source language.

**Release Sequence**

After an atomic operation A performs a release operation on a set of memory locations M, the release sequence headed by A is the longest continuous subsequence of A’s scoped modification order that consists of:

- the atomic operation A as its first element
- atomic read-modify-write operations on M by any agent

The atomics in the last bullet must be mutually-ordered with A by virtue of being in A’s scoped modification order.
Synchronizes-With

Synchronizes-with is a relation between operations, where each operation is either an atomic operation or a memory barrier (aka fence on the host).

If A and B are atomic operations, then A synchronizes-with B if and only if all of the following are true:

- A performs a release operation
- B performs an acquire operation
- A and B are mutually-ordered
- B reads a value written by A or by an operation in the release sequence headed by A

OpControlBarrier, OpMemoryBarrier, and OpMemoryNamedBarrier are memory barrier instructions in SPIR-V.

If A is a release barrier and B is an atomic operation that performs an acquire operation, then A synchronizes-with B if and only if all of the following are true:

- there exists an atomic write X (with any memory semantics)
- A is program-ordered before X
- X and B are mutually-ordered
- B reads a value written by X or by an operation in the release sequence headed by X
  - If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
- A and B are in the instance of each other’s memory scopes
- X’s storage class is in A’s semantics.

If A is an atomic operation that performs a release operation and B is an acquire barrier, then A synchronizes-with B if and only if all of the following are true:

- there exists an atomic read X (with any memory semantics)
- X is program-ordered before B
- X and A are mutually-ordered
- X reads a value written by A or by an operation in the release sequence headed by A
- A and B are in the instance of each other’s memory scopes
- X’s storage class is in B’s semantics.

If A is a release barrier and B is an acquire barrier, then A synchronizes-with B if all of the following are true:
• there exists an atomic write X (with any memory semantics)
• A is program-ordered before X
• there exists an atomic read Y (with any memory semantics)
• Y is program-ordered before B
• X and Y are mutually-ordered
• Y reads the value written by X or by an operation in the release sequence headed by X
  ◦ If X is relaxed, it is still considered to head a hypothetical release sequence for this rule
• A and B are in the instance of each other’s memory scopes
• X’s and Y’s storage class is in A’s and B’s semantics.
  ◦ NOTE: X and Y must have the same storage class, because they are mutually ordered.

If A is a release barrier and B is an acquire barrier and C is a control barrier (where A can optionally equal C and B can optionally equal C), then A synchronizes-with B if all of the following are true:

• A is program-ordered before (or equals) C
• C is program-ordered before (or equals) B
• A and B are in the instance of each other’s memory scopes
• A and B are in the instance of C’s execution scope

Note
This is similar to the barrier-barrier synchronization above, but with a control barrier filling the role of the relaxed atomics.

Let F be an ordering of fragment shader invocations, such that invocation \( F_1 \) is ordered before invocation \( F_2 \) if and only if \( F_1 \) and \( F_2 \) overlap as described in Fragment Shader Interlock and \( F_1 \) executes the interlocked code before \( F_2 \).

If A is an \texttt{OpEndInvocationInterlockEXT} instruction and B is an \texttt{OpBeginInvocationInterlockEXT} instruction, then A synchronizes-with B if the agent that executes A is ordered before the agent that executes B in F. A and B are both considered to have FragmentInterlock memory scope and semantics of UniformMemory and ImageMemory, and A is considered to have Release semantics and B is considered to have Acquire semantics.

Note
\texttt{OpBeginInvocationInterlockEXT} and \texttt{OpBeginInvocationInterlockEXT} do not perform implicit availability or visibility operations. Usually, shaders using fragment shader interlock will declare the relevant resources as coherent to get implicit per-instruction availability and visibility operations.

If A is a release barrier and B is an acquire barrier, then A synchronizes-with B if all of the following are true:
• A is shader-call-ordered-before B
• A and B are in the instance of each other’s memory scopes

No other release and acquire barriers synchronize-with each other.

**System-Synchronizes-With**

System-synchronizes-with is a relation between arbitrary operations on the device or host. Certain operations system-synchronize-with each other, which informally means the first operation occurs before the second and that the synchronization is performed without using application-visible memory accesses.

If there is an execution dependency between two operations A and B, then the operation in the first synchronization scope system-synchronizes-with the operation in the second synchronization scope.

### Note

This covers all Vulkan synchronization primitives, including device operations executing before a synchronization primitive is signaled, wait operations happening before subsequent device operations, signal operations happening before host operations that wait on them, and host operations happening before `vkQueueSubmit`. The list is spread throughout the synchronization chapter, and is not repeated here.

System-synchronizes-with implicitly includes all storage class semantics and has CrossDevice scope.

If A system-synchronizes-with B, we also say A is system-synchronized-before B and B is system-synchronized-after A.

**Private vs. Non-Private**

By default, non-atomic memory operations are treated as private, meaning such a memory operation is not intended to be used for communication with other agents. Memory operations with the NonPrivatePointer/NonPrivateTexel bit set are treated as non-private, and are intended to be used for communication with other agents.

More precisely, for private memory operations to be Location-Ordered between distinct agents requires using system-synchronizes-with rather than shader-based synchronization. Non-private memory operations still obey program-order.

Atomic operations are always considered non-private.

**Inter-Thread-Happens-Before**

Let SC be a non-empty set of storage class semantics. Then (using template syntax) operation A \( \text{inter-thread-happens-before} <\text{SC}> \) operation B if and only if any of the following is true:

• A system-synchronizes-with B
• A synchronizes-with B, and both A and B have all of SC in their semantics
• A is an operation on memory in a storage class in SC or that has all of SC in its semantics, B is a release barrier or release atomic with all of SC in its semantics, and A is program-ordered before B
• A is an acquire barrier or acquire atomic with all of SC in its semantics, B is an operation on memory in a storage class in SC or that has all of SC in its semantics, and A is program-ordered before B
• A and B are both host operations and A inter-thread-happens-before B as defined in the host language spec
• A inter-thread-happens-before<SC> some X and X inter-thread-happens-before<SC> B

Happens-Before

Operation A happens-before operation B if and only if any of the following is true:

• A is program-ordered before B
• A inter-thread-happens-before<SC> B for some set of storage classes SC

Happens-after is defined similarly.

Note
Unlike C++, happens-before is not always sufficient for a write to be visible to a read. Additional availability and visibility operations may be required for writes to be visible-to other memory accesses.

Note
Happens-before is not transitive, but each of program-order and inter-thread-happens-before<SC> are transitive. These can be thought of as covering the “single-threaded” case and the “multi-threaded” case, and it is not necessary (and not valid) to form chains between the two.

Availability and Visibility

Availability and visibility are states of a write operation, which (informally) track how far the write has permeated the system, i.e. which agents and references are able to observe the write. Availability state is per memory domain. Visibility state is per (agent,reference) pair. Availability and visibility states are per-memory location for each write.

Memory domains are named according to the agents whose memory accesses use the domain. Domains used by shader invocations are organized hierarchically into multiple smaller memory domains which correspond to the different scopes. Each memory domain is considered the dual of a scope, and vice versa. The memory domains defined in Vulkan include:

• host - accessible by host agents
• device - accessible by all device agents for a particular device
• shader - accessible by shader agents for a particular device, corresponding to the Device scope.

• queue family instance - accessible by shader agents in a single queue family, corresponding to the QueueFamily scope.

• fragment interlock instance - accessible by fragment shader agents that overlap, corresponding to the FragmentInterlock scope.

• shader call instance - accessible by shader agents that are shader-call-related, corresponding to the ShaderCallKHR scope.

• workgroup instance - accessible by shader agents in the same workgroup, corresponding to the Workgroup scope.

• subgroup instance - accessible by shader agents in the same subgroup, corresponding to the Subgroup scope.

The memory domains are nested in the order listed above, except for shader call instance domain, with memory domains later in the list nested in the domains earlier in the list. The shader call instance domain is at an implementation-dependent location in the list, and is nested according to that location. The shader call instance domain is not broader than the queue family instance domain.

Note
Memory domains do not correspond to storage classes or device-local and host-local VkDeviceMemory allocations, rather they indicate whether a write can be made visible only to agents in the same subgroup, same workgroup, overlapping fragment shader invocation, shader-call-related ray tracing invocation, in any shader invocation, or anywhere on the device, or host. The shader, queue family instance, fragment interlock instance, shader call instance, workgroup instance, and subgroup instance domains are only used for shader-based availability/visibility operations, in other cases writes can be made available from/visible to the shader via the device domain.

Availability operations, visibility operations, and memory domain operations alter the state of the write operations that happen-before them, and which are included in their source scope to be available or visible to their destination scope.

• For an availability operation, the source scope is a set of (agent,reference,memory location) tuples, and the destination scope is a set of memory domains.

• For a memory domain operation, the source scope is a memory domain and the destination scope is a memory domain.

• For a visibility operation, the source scope is a set of memory domains and the destination scope is a set of (agent,reference,memory location) tuples.

How the scopes are determined depends on the specific operation. Availability and memory domain operations expand the set of memory domains to which the write is available. Visibility operations expand the set of (agent,reference,memory location) tuples to which the write is visible.

Recall that availability and visibility states are per-memory location, and let W be a write operation to one or more locations performed by agent A via reference R. Let L be one of the locations...
written. \((W,L)\) (the write \(W\) to \(L\)), is initially not available to any memory domain and only visible to \((A,R,L)\). An availability operation \(AV\) that happens-after \(W\) and that includes \((A,R,L)\) in its source scope makes \((W,L)\) available to the memory domains in its destination scope.

A memory domain operation \(DOM\) that happens-after \(AV\) and for which \((W,L)\) is available in the source scope makes \((W,L)\) available in the destination memory domain.

A visibility operation \(VIS\) that happens-after \(AV\) (or \(DOM\)) and for which \((W,L)\) is available in any domain in the source scope makes \((W,L)\) visible to all \((agent,reference,L)\) tuples included in its destination scope.

If write \(W_2\) happens-after \(W\), and their sets of memory locations overlap, then \(W\) will not be available/visible to all agents/references for those memory locations that overlap (and future \(AV/DOM/VIS\) ops cannot revive \(W\)'s write to those locations).

Availability, memory domain, and visibility operations are treated like other non-atomic memory accesses for the purpose of memory semantics, meaning they can be ordered by release-acquire sequences or memory barriers.

An availability chain is a sequence of availability operations to increasingly broad memory domains, where element \(N+1\) of the chain is performed in the dual scope instance of the destination memory domain of element \(N\) and element \(N\) happens-before element \(N+1\). An example is an availability operation with destination scope of the workgroup instance domain that happens-before an availability operation to the shader domain performed by an invocation in the same workgroup. An availability chain \(AVC\) that happens-after \(W\) and that includes \((A,R,L)\) in the source scope makes \((W,L)\) available to the memory domains in its final destination scope. An availability chain with a single element is just the availability operation.

Similarly, a visibility chain is a sequence of visibility operations from increasingly narrow memory domains, where element \(N\) of the chain is performed in the dual scope instance of the source memory domain of element \(N+1\) and element \(N\) happens-before element \(N+1\). An example is a visibility operation with source scope of the shader domain that happens-before a visibility operation with source scope of the workgroup instance domain performed by an invocation in the same workgroup. A visibility chain \(VISC\) that happens-after \(AVC\) (or \(DOM\)) and for which \((W,L)\) is available in any domain in the source scope makes \((W,L)\) visible to all \((agent,reference,L)\) tuples included in its final destination scope. A visibility chain with a single element is just the visibility operation.

### Availability, Visibility, and Domain Operations

The following operations generate availability, visibility, and domain operations. When multiple availability/visibility/domain operations are described, they are system-synchronized-with each other in the order listed.

An operation that performs a memory dependency generates:

- If the source access mask includes \(VK_ACCESS_HOST_WRITE_BIT\), then the dependency includes a memory domain operation from host domain to device domain.
- An availability operation with source scope of all writes in the first access scope of the
dependency and a destination scope of the device domain.

- A visibility operation with source scope of the device domain and destination scope of the second access scope of the dependency.

- If the destination access mask includes `VK_ACCESS_HOST_READ_BIT` or `VK_ACCESS_HOST_WRITE_BIT`, then the dependency includes a memory domain operation from device domain to host domain.

`vkFlushMappedMemoryRanges` performs an availability operation, with a source scope of 
(agents,references) = (all host threads, all mapped memory ranges passed to the command), and destination scope of the host domain.

`vkInvalidateMappedMemoryRanges` performs a visibility operation, with a source scope of the host domain and a destination scope of 
(agents,references) = (all host threads, all mapped memory ranges passed to the command).

`vkQueueSubmit` performs a memory domain operation from host to device, and a visibility operation with source scope of the device domain and destination scope of all agents and references on the device.

### Availability and Visibility Semantics

A memory barrier or atomic operation via agent A that includes MakeAvailable in its semantics performs an availability operation whose source scope includes agent A and all references in the storage classes in that instruction’s storage class semantics, and all memory locations, and whose destination scope is a set of memory domains selected as specified below. The implicit availability operation is program-ordered between the barrier or atomic and all other operations program-ordered before the barrier or atomic.

A memory barrier or atomic operation via agent A that includes MakeVisible in its semantics performs a visibility operation whose source scope is a set of memory domains selected as specified below, and whose destination scope includes agent A and all references in the storage classes in that instruction’s storage class semantics, and all memory locations. The implicit visibility operation is program-ordered between the barrier or atomic and all other operations program-ordered after the barrier or atomic.

The memory domains are selected based on the memory scope of the instruction as follows:

- **Device** scope uses the shader domain
- **QueueFamily** scope uses the queue family instance domain
- **FragmentInterlock** scope uses the fragment interlock instance domain
- **ShaderCallKHR** scope uses the shader call instance domain
- **Workgroup** scope uses the workgroup instance domain
- **Subgroup** uses the subgroup instance domain
- **Invocation** perform no availability/visibility operations.

When an availability operation performed by an agent A includes a memory domain D in its destination scope, where D corresponds to scope instance S, it also includes the memory domains
that correspond to each smaller scope instance \( S' \) that is a subset of \( S \) and that includes \( A \). Similarly for visibility operations.

## Per-Instruction Availability and Visibility Semantics

A memory write instruction that includes `MakePointerAvailable`, or an image write instruction that includes `MakeTexelAvailable`, performs an availability operation whose source scope includes the agent and reference used to perform the write and the memory locations written by the instruction, and whose destination scope is a set of memory domains selected by the `Scope` operand specified in *Availability and Visibility Semantics*. The implicit availability operation is program-ordered between the write and all other operations program-ordered after the write.

A memory read instruction that includes `MakePointerVisible`, or an image read instruction that includes `MakeTexelVisible`, performs a visibility operation whose source scope is a set of memory domains selected by the `Scope` operand as specified in *Availability and Visibility Semantics*, and whose destination scope includes the agent and reference used to perform the read and the memory locations read by the instruction. The implicit visibility operation is program-ordered between read and all other operations program-ordered before the read.

**Note**

Although reads with per-instruction visibility only perform visibility ops from the shader or fragment interlock instance or shader call instance or workgroup instance or subgroup instance domain, they will also see writes that were made visible via the device domain, i.e. those writes previously performed by non-shader agents and made visible via API commands.

**Note**

It is expected that all invocations in a subgroup execute on the same processor with the same path to memory, and thus availability and visibility operations with subgroup scope can be expected to be “free”.

## Location-Ordered

Let \( X \) and \( Y \) be memory accesses to overlapping sets of memory locations \( M \), where \( X \neq Y \). Let \((A_X, R_X)\) be the agent and reference used for \( X \), and \((A_Y, R_Y)\) be the agent and reference used for \( Y \). For now, let “\( \rightarrow \)” denote happens-before and “\( \rightarrow_{rcpo} \)” denote the reflexive closure of program-ordered before.

If \( D_1 \) and \( D_2 \) are different memory domains, then let \( \text{DOM}(D_1, D_2) \) be a memory domain operation from \( D_1 \) to \( D_2 \). Otherwise, let \( \text{DOM}(D, D) \) be a placeholder such that \( X \rightarrow \text{DOM}(D, D) \rightarrow Y \) if and only if \( X \rightarrow Y \).

\( X \) is *location-ordered* before \( Y \) for a location \( L \) in \( M \) if and only if any of the following is true:

- \( A_X = A_Y \) and \( R_X = R_Y \) and \( X \rightarrow Y \)
  
  **Note**: this case means no availability/visibility ops are required when it is the same (agent,reference).
• X is a read, both X and Y are non-private, and X → Y

• X is a read, and X (transitively) system-synchronizes with Y

• If \( R_X = R_Y \) and \( A_X \) and \( A_Y \) access a common memory domain \( D \) (e.g. are in the same workgroup instance if \( D \) is the workgroup instance domain), and both X and Y are non-private:
  ◦ X is a write, Y is a write, \( AVC(A_X,R_X,D,L) \) is an availability chain making \( (X,L) \) available to domain D, and \( X \rightarrow^{\text{rpo}} AVC(A_X,R_X,D,L) \rightarrow Y \)
  ◦ X is a write, Y is a read, \( AVC(A_X,R_X,D,L) \) is an availability chain making \( (X,L) \) available to domain D, \( VIS(A_Y,R_Y,D,L) \) is a visibility chain making writes to \( L \) available in domain \( D \) visible to \( Y \), and \( X \rightarrow^{\text{rpo}} AVC(A_X,R_X,D,L) \rightarrow VIS(A_Y,R_Y,D,L) \rightarrow^{\text{rpo}} Y \)
  ◦ If \( \text{VkPhysicalDeviceVulkanMemoryModelFeatures}::\text{vulkanMemoryModelAvailabilityVisibilityChains} = \text{VK_FALSE} \), then AVC and VISC must each only have a single element in the chain, in each sub-bullet above.

• Let \( D_X \) and \( D_Y \) each be either the device domain or the host domain, depending on whether \( A_X \) and \( A_Y \) execute on the device or host:
  ◦ X is a write and Y is a write, and \( X \rightarrow AV(A_X,R_X,D_X,L) \rightarrow DOM(D_X,D_Y) \rightarrow Y \)
  ◦ X is a write and Y is a read, and \( X \rightarrow AV(A_X,R_X,D_X,L) \rightarrow DOM(D_X,D_Y) \rightarrow VIS(A_Y,R_Y,D_Y,L) \rightarrow Y \)

Note
The final bullet (synchronization through device/host domain) requires API-level synchronization operations, since the device/host domains are not accessible via shader instructions. And “device domain” is not to be confused with “device scope”, which synchronizes through the “shader domain”.

Data Race

Let X and Y be operations that access overlapping sets of memory locations \( M \), where X ≠ Y, and at least one of X and Y is a write, and X and Y are not mutually-ordered atomic operations. If there does not exist a location-ordered relation between X and Y for each location in \( M \), then there is a data race.

Applications must ensure that no data races occur during the execution of their application.

Note
Data races can only occur due to instructions that are actually executed. For example, an instruction skipped due to control flow must not contribute to a data race.

Visible-To

Let X be a write and Y be a read whose sets of memory locations overlap, and let \( M \) be the set of memory locations that overlap. Let \( M_1 \) be a non-empty subset of \( M \). Then X is visible-to Y for memory locations \( M_2 \) if and only if all of the following are true:
• X is location-ordered before Y for each location L in M₂.

• There does not exist another write Z to any location L in M₂ such that X is location-ordered before Z for location L and Z is location-ordered before Y for location L.

If X is visible-to Y, then Y reads the value written by X for locations M₂.

**Note**
It is possible for there to be a write between X and Y that overwrites a subset of the memory locations, but the remaining memory locations (M₂) will still be visible-to Y.

**Acyclicity**

*Reads-from* is a relation between operations, where the first operation is a write, the second operation is a read, and the second operation reads the value written by the first operation. *From-reads* is a relation between operations, where the first operation is a read, the second operation is a write, and the first operation reads a value written earlier than the second operation in the second operation’s scoped modification order (or the first operation reads from the initial value, and the second operation is any write to the same locations).

Then the implementation must guarantee that no cycles exist in the union of the following relations:

• location-ordered
• scoped modification order (over all atomic writes)
• reads-from
• from-reads

**Note**
This is a “consistency” axiom, which informally guarantees that sequences of operations can’t violate causality.

**Scoped Modification Order Coherence**

Let A and B be mutually-ordered atomic operations, where A is location-ordered before B. Then the following rules are a consequence of acyclicity:

• If A and B are both reads and A does not read the initial value, then the write that A takes its value from must be earlier in its own scoped modification order than (or the same as) the write that B takes its value from (no cycles between location-order, reads-from, and from-reads).

• If A is a read and B is a write and A does not read the initial value, then A must take its value from a write earlier than B in B’s scoped modification order (no cycles between location-order, scope modification order, and reads-from).

• If A is a write and B is a read, then B must take its value from A or a write later than A in A’s scoped modification order (no cycles between location-order, scoped modification order, and...
from-reads).

- If A and B are both writes, then A must be earlier than B in A's scoped modification order (no cycles between location-order and scoped modification order).
- If A is a write and B is a read-modify-write and B reads the value written by A, then B comes immediately after A in A's scoped modification order (no cycles between scoped modification order and from-reads).

**Shader I/O**

If a shader invocation A in a shader stage other than Vertex performs a memory read operation X from an object in storage class CallableDataKHR, IncomingCallableDataKHR, RayPayloadKHR, HitAttributeKHR, IncomingRayPayloadKHR, or Input, then X is system-synchronized-after all writes to the corresponding CallableDataKHR, IncomingCallableDataKHR, RayPayloadKHR, HitAttributeKHR, IncomingRayPayloadKHR, or Output storage variable(s) in the shader invocation(s) that contribute to generating invocation A, and those writes are all visible-to X.

**Note**

It is not necessary for the upstream shader invocations to have completed execution, they only need to have generated the output that is being read.

**Deallocation**

A call to vkFreeMemory must happen-after all memory operations on all memory locations in that VkDeviceMemory object.

**Note**

Normally, device memory operations in a given queue are synchronized with vkFreeMemory by having a host thread wait on a fence signalled by that queue, and the wait happens-before the call to vkFreeMemory on the host.

The deallocation of SPIR-V variables is managed by the system and happens-after all operations on those variables.

**Descriptions (Informative)**

This subsection offers more easily understandable consequences of the memory model for app/compiler developers.

Let SC be the storage class(es) specified by a release or acquire operation or barrier.

- An atomic write with release semantics must not be reordered against any read or write to SC that is program-ordered before it (regardless of the storage class the atomic is in).
- An atomic read with acquire semantics must not be reordered against any read or write to SC that is program-ordered after it (regardless of the storage class the atomic is in).
- Any write to SC program-ordered after a release barrier must not be reordered against any read
or write to SC program-ordered before that barrier.

- Any read from SC program-ordered before an acquire barrier must not be reordered against any read or write to SC program-ordered after the barrier.

A control barrier (even if it has no memory semantics) must not be reordered against any memory barriers.

This memory model allows memory accesses with and without availability and visibility operations, as well as atomic operations, all to be performed on the same memory location. This is critical to allow it to reason about memory that is reused in multiple ways, e.g. across the lifetime of different shader invocations or draw calls. While GLSL (and legacy SPIR-V) applies the “coherent” decoration to variables (for historical reasons), this model treats each memory access instruction as having optional implicit availability/visibility operations. GLSL to SPIR-V compilers should map all (non-atomic) operations on a coherent variable to Make{Pointer,Texel}{Available}{Visible} flags in this model.

Atomic operations implicitly have availability/visibility operations, and the scope of those operations is taken from the atomic operation’s scope.

**Tessellation Output Ordering**

For SPIR-V that uses the Vulkan Memory Model, the `OutputMemory` storage class is used to synchronize accesses to tessellation control output variables. For legacy SPIR-V that does not enable the Vulkan Memory Model via `OpMemoryModel`, tessellation outputs can be ordered using a control barrier with no particular memory scope or semantics, as defined below.

Let X and Y be memory operations performed by shader invocations $A_X$ and $A_Y$. Operation X is **tessellation-output-ordered** before operation Y if and only if all of the following are true:

- There is a dynamic instance of an `OpControlBarrier` instruction C such that X is program-ordered before C in $A_X$ and C is program-ordered before Y in $A_Y$.
- $A_X$ and $A_Y$ are in the same instance of C’s execution scope.

If shader invocations $A_X$ and $A_Y$ in the TessellationControl execution model execute memory operations X and Y, respectively, on the `Output` storage class, and X is tessellation-output-ordered before Y with a scope of Workgroup, then X is location-ordered before Y, and if X is a write and Y is a read then X is visible-to Y.

**Cooperative Matrix Memory Access**

For each dynamic instance of a cooperative matrix load or store instruction (`OpCooperativeMatrixLoadNV` or `OpCooperativeMatrixStoreNV`), a single implementation-dependent invocation within the instance of the matrix’s scope performs a non-atomic load or store (respectively) to each memory location that is defined to be accessed by the instruction.
Appendix C: Compressed Image Formats

The compressed texture formats used by Vulkan are described in the specifically identified sections of the Khronos Data Format Specification, version 1.3.

Unless otherwise described, the quantities encoded in these compressed formats are treated as normalized, unsigned values.

Those formats listed as sRGB-encoded have in-memory representations of R, G and B components which are nonlinearly-encoded as R', G', and B'; any alpha component is unchanged. As part of filtering, the nonlinear R', G', and B' values are converted to linear R, G, and B components; any alpha component is unchanged. The conversion between linear and nonlinear encoding is performed as described in the “KHR_DF_TRANSFER_SRGB” section of the Khronos Data Format Specification.
Block-Compressed Image Formats

BC1, BC2 and BC3 formats are described in “S3TC Compressed Texture Image Formats” chapter of the Khronos Data Format Specification. BC4 and BC5 are described in the “RGTC Compressed Texture Image Formats” chapter. BC6H and BC7 are described in the “BPTC Compressed Texture Image Formats” chapter.

Table 88. Mapping of Vulkan BC formats to descriptions

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Khronos Data Format Specification description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formats described in the “S3TC Compressed Texture Image Formats” chapter</strong></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
<td>BC1 with no alpha</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
<td>BC1 with no alpha, sRGB-encoded</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
<td>BC1 with alpha</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td>BC1 with alpha, sRGB-encoded</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
<td>BC2</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_SRGB_BLOCK</td>
<td>BC2, sRGB-encoded</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
<td>BC3</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_SRGB_BLOCK</td>
<td>BC3, sRGB-encoded</td>
</tr>
<tr>
<td><strong>Formats described in the “RGTC Compressed Texture Image Formats” chapter</strong></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_BLOCK</td>
<td>BC4 unsigned</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
<td>BC4 signed</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
<td>BC5 unsigned</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
<td>BC5 signed</td>
</tr>
<tr>
<td><strong>Formats described in the “BPTC Compressed Texture Image Formats” chapter</strong></td>
<td></td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK</td>
<td>BC6H (unsigned version)</td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
<td>BC6H (signed version)</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
<td>BC7</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_SRGB_BLOCK</td>
<td>BC7, sRGB-encoded</td>
</tr>
</tbody>
</table>
ETC Compressed Image Formats

The following formats are described in the “ETC2 Compressed Texture Image Formats” chapter of the Khronos Data Format Specification.

**Table 89. Mapping of Vulkan ETC formats to descriptions**

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Khronos Data Format Specification description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK</td>
<td>RGB ETC2</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</td>
<td>RGB ETC2 with sRGB encoding</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK</td>
<td>RGB ETC2 with punch-through alpha</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</td>
<td>RGB ETC2 with punch-through alpha and sRGB</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK</td>
<td>RGBA ETC2</td>
</tr>
<tr>
<td>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</td>
<td>RGBA ETC2 with sRGB encoding</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_UNORM_BLOCK</td>
<td>Unsigned R11 EAC</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11_SNORM_BLOCK</td>
<td>Signed R11 EAC</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_UNORM_BLOCK</td>
<td>Unsigned RG11 EAC</td>
</tr>
<tr>
<td>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</td>
<td>Signed RG11 EAC</td>
</tr>
</tbody>
</table>
ASTC Compressed Image Formats

ASTC formats are described in the “ASTC Compressed Texture Image Formats” chapter of the Khronos Data Format Specification.

Table 90. Mapping of Vulkan ASTC formats to descriptions

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Compressed texel block dimensions</th>
<th>sRGB-encoded</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</td>
<td>4 × 4</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>4 × 4</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>5 × 4</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>5 × 4</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>5 × 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>5 × 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>6 × 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>6 × 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>6 × 6</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>6 × 6</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>8 × 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>8 × 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>8 × 6</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>8 × 6</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>8 × 8</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>8 × 8</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>10 × 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>10 × 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>10 × 6</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>10 × 6</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>10 × 8</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
<td>10 × 8</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</td>
<td>10 × 10</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
<td>10 × 10</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK</td>
<td>12 × 10</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
<td>12 × 10</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK</td>
<td>12 × 12</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VkFormat</td>
<td>Compresed texel block dimensions</td>
<td>sRGB-encoded</td>
<td>Profile</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------</td>
<td>--------------</td>
<td>---------</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_SRGB_BLOCK</td>
<td>12 × 12</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT</td>
<td>4 × 4</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT</td>
<td>5 × 4</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT</td>
<td>5 × 5</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT</td>
<td>6 × 5</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT</td>
<td>6 × 6</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SFLOAT_BLOCK_EXT</td>
<td>8 × 5</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SFLOAT_BLOCK_EXT</td>
<td>8 × 6</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SFLOAT_BLOCK_EXT</td>
<td>8 × 8</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT</td>
<td>10 × 5</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT</td>
<td>10 × 6</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT</td>
<td>10 × 8</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT</td>
<td>10 × 10</td>
<td>No</td>
<td>HDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT</td>
<td>12 × 12</td>
<td>No</td>
<td>HDR</td>
</tr>
</tbody>
</table>

**ASTC decode mode**

If the `VK_EXT_astc_decode_mode` extension is enabled, the decode mode is determined as follows:

Table 91. Mapping of Vulkan ASTC decoding format to ASTC decoding modes

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Decoding mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
<td>decode_float16</td>
</tr>
<tr>
<td>VK_FORMAT_R8G8B8A8_UNORM</td>
<td>decode_unorm8</td>
</tr>
<tr>
<td>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</td>
<td>decode_rgb9e5</td>
</tr>
</tbody>
</table>

Otherwise, the ASTC decode mode is decode_float16.
PVRTC Compressed Image Formats

PVRTC formats are described in the “PVRTC Compressed Texture Image Formats” chapter of the Khronos Data Format Specification.

Table 92. Mapping of Vulkan PVRTC formats to descriptions

<table>
<thead>
<tr>
<th>VkFormat</th>
<th>Compressed texel block dimensions</th>
<th>sRGB-encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG</td>
<td>8 × 4</td>
<td>No</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG</td>
<td>4 × 4</td>
<td>No</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG</td>
<td>8 × 4</td>
<td>No</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG</td>
<td>4 × 4</td>
<td>No</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG</td>
<td>8 × 4</td>
<td>Yes</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG</td>
<td>4 × 4</td>
<td>Yes</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG</td>
<td>8 × 4</td>
<td>Yes</td>
</tr>
<tr>
<td>VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG</td>
<td>4 × 4</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Appendix D: Core Revisions (Informative)

New minor versions of the Vulkan API are defined periodically by the Khronos Vulkan Working Group. These consist of some amount of additional functionality added to the core API, potentially including both new functionality and functionality promoted from extensions.

It is possible to build the specification for earlier versions, but to aid readability of the latest versions, this appendix gives an overview of the changes as compared to earlier versions.

Version 1.1

Vulkan Version 1.1 promoted a number of key extensions into the core API:

- VK_KHR_16bit_storage
- VK_KHR_bind_memory2
- VK_KHR_dedicated_allocation
- VK_KHR_descriptor_update_template
- VK_KHR_device_group
- VK_KHR_device_group_creation
- VK_KHR_external_fence
- VK_KHR_external_fence_capabilities
- VK_KHR_external_memory
- VK_KHR_external_memory_capabilities
- VK_KHR_external_semaphore
- VK_KHR_external_semaphore_capabilities
- VK_KHR_get_memory_requirements2
- VK_KHR_get_physical_device_properties2
- VK_KHR_maintenance1
- VK_KHR_maintenance2
- VK_KHR_maintenance3
- VK_KHR_multiview
- VK_KHR_relaxed_block_layout
- VK_KHR_sampler_ycbcr_conversion
- VK_KHR_shader_draw_parameters
- VK_KHR_storage_buffer_storage_class
- VK_KHR_variable_pointers
All differences in behavior between these extensions and the corresponding Vulkan 1.1 functionality are summarized below.

**Differences relative to **VK_KHR_16bit_storage**

If the VK_KHR_16bit_storage extension is not supported, support for the storageBuffer16BitAccess feature is optional. Support for this feature is defined by VkPhysicalDevice16BitStorageFeatures::storageBuffer16BitAccess when queried via vkGetPhysicalDeviceFeatures2.

**Differences relative to **VK_KHR_sampler_ycbcr_conversion**

If the VK_KHR_sampler_ycbcr_conversion extension is not supported, support for the samplerYcbcrConversion feature is optional. Support for this feature is defined by VkPhysicalDeviceSamplerYcbcrConversionFeatures::samplerYcbcrConversion when queried via vkGetPhysicalDeviceFeatures2.

**Differences relative to **VK_KHR_shader_draw_parameters**

If the VK_KHR_shader_draw_parameters extension is not supported, support for the SPV_KHR_shader_draw_parameters SPIR-V extension is optional. Support for this feature is defined by VkPhysicalDeviceShaderDrawParametersFeatures::shaderDrawParameters when queried via vkGetPhysicalDeviceFeatures2.

**Differences relative to **VK_KHR_variable_pointers**

If the VK_KHR_variable_pointers extension is not supported, support for the variablePointersStorageBuffer feature is optional. Support for this feature is defined by VkPhysicalDeviceVariablePointersFeatures::variablePointersStorageBuffer when queried via vkGetPhysicalDeviceFeatures2.

**Additional Vulkan 1.1 Feature Support**

In addition to the promoted extensions described above, Vulkan 1.1 added support for:

- The group operations and subgroup scope.
- The protected memory feature.
- A new command to enumerate the instance version: vkEnumerateInstanceVersion.
- The VkPhysicalDeviceShaderDrawParametersFeatures feature query struct (where the VK_KHR_shader_draw_parameters extension did not have one).

**New Macros**

- VK_API_VERSION_1_1

**New Object Types**

- VkDescriptorUpdateTemplate
• VkSamplerYcbcrConversion

New Commands

• vkBindBufferMemory2
• vkBindImageMemory2
• vkCmdDispatchBase
• vkCmdSetDeviceMask
• vkCreateDescriptorUpdateTemplate
• vkCreateSamplerYcbcrConversion
• vkDestroyDescriptorUpdateTemplate
• vkDestroySamplerYcbcrConversion
• vkEnumerateInstanceVersion
• vkEnumeratePhysicalDeviceGroups
• vkGetBufferMemoryRequirements2
• vkGetDescriptorSetLayoutSupport
• vkGetDeviceGroupPeerMemoryFeatures
• vkGetDeviceQueue2
• vkGetImageMemoryRequirements2
• vkGetImageSparseMemoryRequirements2
• vkGetPhysicalDeviceExternalBufferProperties
• vkGetPhysicalDeviceExternalFenceProperties
• vkGetPhysicalDeviceExternalSemaphoreProperties
• vkGetPhysicalDeviceFeatures2
• vkGetPhysicalDeviceFormatProperties2
• vkGetPhysicalDeviceImageFormatProperties2
• vkGetPhysicalDeviceMemoryProperties2
• vkGetPhysicalDeviceQueueFamilyProperties2
• vkGetPhysicalDeviceSparseImageFormatProperties2
• vkTrimCommandPool
• vkUpdateDescriptorSetWithTemplate

New Structures

• VkBindBufferMemoryInfo
• VkBindImageMemoryInfo
• VkBufferMemoryRequirementsInfo2
• VkDescriptorSetLayoutSupport
• VkDescriptorUpdateTemplateCreateInfo
• VkDescriptorUpdateTemplateEntry
• VkDeviceQueueInfo2
• VkExternalBufferProperties
• VkExternalFenceProperties
•VkExternalMemoryProperties
• VkExternalSemaphoreProperties
• VkFormatProperties2
• VkImageFormatProperties2
• VkImageMemoryRequirementsInfo2
• VkImageSparseMemoryRequirementsInfo2
• VkInputAttachmentAspectReference
• VkMemoryRequirements2
• VkPhysicalDeviceExternalBufferInfo
• VkPhysicalDeviceExternalFenceInfo
• VkPhysicalDeviceExternalSemaphoreInfo
• VkPhysicalDeviceGroupProperties
• VkPhysicalDeviceImageFormatInfo2
• VkPhysicalDeviceMemoryProperties2
• VkPhysicalDeviceProperties2
• VkPhysicalDeviceSparseImageFormatInfo2
• VkQueueFamilyProperties2
• VkSamplerYcbcrConversionCreateInfo
• VkSparseImageFormatProperties2
• VkSparseImageMemoryRequirements2
• Extending VkBindBufferMemoryInfo:
  ◦ VkBindBufferMemoryDeviceGroupInfo
• Extending VkBindImageMemoryInfo:
  ◦ VkBindImageMemoryDeviceGroupInfo
  ◦ VkBindImagePlaneMemoryInfo
• Extending VkBindSparseInfo:
  ◦ VkDeviceGroupBindSparseInfo
• Extending VkBufferCreateInfo:
• `VkExternalMemoryBufferCreateInfo`

• **Extending** `VkCommandBufferBeginInfo`:
  • `VkDeviceGroupCommandBufferBeginInfo`

• **Extending** `VkDeviceCreateInfo`:
  • `VkDeviceGroupDeviceCreateInfo`
  • `VkPhysicalDeviceFeatures2`

• **Extending** `VkFenceCreateInfo`:
  • `VkExportFenceCreateInfo`

• **Extending** `VkImageCreateInfo`:
  • `VkExternalMemoryImageCreateInfo`
  • `VkImageFormatProperties2`:
    • `VkExternalImageFormatProperties`
    • `VkSamplerYcbcrConversionImageFormatProperties`

• **Extending** `VkImageMemoryRequirementsInfo2`:
  • `VkImagePlaneMemoryRequirementsInfo`

• **Extending** `VkImageViewCreateInfo`:
  • `VkImageViewUsageCreateInfo`

• **Extending** `VkMemoryAllocateInfo`:
  • `VkExportMemoryAllocateInfo`
  • `VkMemoryAllocateFlagsInfo`
  • `VkMemoryDedicatedAllocateInfo`

• **Extending** `VkMemoryRequirements2`:
  • `VkMemoryDedicatedRequirements`

• **Extending** `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  • `VkPhysicalDevice16BitStorageFeatures`
  • `VkPhysicalDeviceMultiviewFeatures`
  • `VkPhysicalDeviceProtectedMemoryFeatures`
  • `VkPhysicalDeviceSamplerYcbcrConversionFeatures`
  • `VkPhysicalDeviceShaderDrawParametersFeatures`
  • `VkPhysicalDeviceVariablePointersFeatures`

• **Extending** `VkPhysicalDeviceImageFormatInfo2`:
  • `VkPhysicalDeviceExternalImageFormatInfo`

• **Extending** `VkPhysicalDeviceProperties2`:
  • `VkPhysicalDeviceIDProperties`
  • `VkPhysicalDeviceMaintenance3Properties`
• VkPhysicalDeviceMultiviewProperties
• VkPhysicalDevicePointClippingProperties
• VkPhysicalDeviceProtectedMemoryProperties
• VkPhysicalDeviceSubgroupProperties
  • Extending VkPipelineTessellationStateCreateInfo:
    ◦ VkPipelineTessellationDomainOriginStateCreateInfo
  • Extending VkRenderPassBeginInfo:
    ◦ VkDeviceGroupRenderPassBeginInfo
  • Extending VkRenderPassCreateInfo:
    ◦ VkRenderPassInputAttachmentAspectCreateInfo
    ◦ VkRenderPassMultiviewCreateInfo
  • Extending VkSamplerCreateInfo, VkImageViewCreateInfo:
    ◦ VkSamplerYcbcrConversionInfo
  • Extending VkSemaphoreCreateInfo:
    ◦ VkExportSemaphoreCreateInfo
  • Extending VkSubmitInfo:
    ◦ VkDeviceGroupSubmitInfo
    ◦ VkProtectedSubmitInfo

**New Enums**

• VkChromaLocation
• VkDescriptorUpdateTemplateType
• VkExternalFenceFeatureFlagBits
• VkExternalFenceHandleTypeFlagBits
• VkExternalMemoryFeatureFlagBits
• VkExternalMemoryHandleTypeFlagBits
• VkExternalSemaphoreFeatureFlagBits
• VkExternalSemaphoreHandleTypeFlagBits
• VkFenceImportFlagBits
• VkMemoryAllocateFlagBits
• VkPeerMemoryFeatureFlagBits
• VkPointClippingBehavior
• VkSamplerYcbcrModelConversion
• VkSamplerYcbcrRange
• VkSemaphoreImportFlagBits
• VkSemaphoreImportFlagBits
New Bitmasks

- VkCommandPoolTrimFlags
- VkDescriptorUpdateTemplateCreateFlags
- VkExternalFenceFeatureFlags
- VkExternalFenceHandleTypeFlags
- VkExternalMemoryFeatureFlags
- VkExternalMemoryHandleTypeFlags
- VkExternalSemaphoreFeatureFlags
- VkExternalSemaphoreHandleTypeFlags
- VkFenceImportFlags
- VkMemoryAllocateFlags
- VkPeerMemoryFeatureFlags
- VkSemaphoreImportFlags
- VkSubgroupFeatureFlags

New Enum Constants

- VK_LUID_SIZE
- VK_MAX_DEVICE_GROUP_SIZE
- VK_QUEUE_FAMILY_EXTERNAL

Extending VkBufferCreateFlagBits:
  - VK_BUFFER_CREATE_PROTECTED_BIT

Extending VkCommandPoolCreateFlagBits:
  - VK_COMMAND_POOL_CREATE_PROTECTED_BIT

Extending VkDependencyFlagBits:
  - VK_DEPENDENCY_DEVICE_GROUP_BIT
  - VK_DEPENDENCY_VIEW_LOCAL_BIT

Extending VkDeviceQueueCreateFlagBits:
  - VKDEVICE_QUEUE_CREATE_PROTECTED_BIT

Extending VkFormat:
  - VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16
  - VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16
  - VK_FORMAT_B16G16R16G16_422_UNORM
- **VK_FORMAT_B8G8R8G8_422_UNORM**
- **VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16**
- **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16**
- **VK_FORMAT_G10X6_B10X6R10X6_2PLANE_422_UNORM_3PACK16**
- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16**
- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16**
- **VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16**
- **VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16**
- **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16**
- **VK_FORMAT_G12X4_B12X4R12X4_2PLANE_422_UNORM_3PACK16**
- **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16**
- **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16**
- **VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16**
- **VK_FORMAT_G16B16G16R16_422_UNORM**
- **VK_FORMAT_G16_B16R16_2PLANE_420_UNORM**
- **VK_FORMAT_G16_B16R16_2PLANE_422_UNORM**
- **VK_FORMAT_G16_B16_R16_3PLANE_420_UNORM**
- **VK_FORMAT_G16_B16_R16_3PLANE_422_UNORM**
- **VK_FORMAT_G16_B16_R16_3PLANE_444_UNORM**
- **VK_FORMAT_G8B8G8R8_422_UNORM**
- **VK_FORMAT_G8_B8R8_2PLANE_420_UNORM**
- **VK_FORMAT_G8_B8R8_2PLANE_422_UNORM**
- **VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM**
- **VK_FORMAT_G8_B8_R8_3PLANE_422_UNORM**
- **VK_FORMAT_G8_B8_R8_3PLANE_444_UNORM**
- **VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_4PACK16**
- **VK_FORMAT_R10X6G10X6B10X6A10X6_422_UNORM_4PACK16**
- **VK_FORMAT_R10X6G10X6B10X6A10X6_2PACK16**
- **VK_FORMAT_R10X6G10X6B10X6A10X6_UNORM_PACK16**
- **VK_FORMAT_R12X4G12X4B12X4A12X4_4PACK16**
- **VK_FORMAT_R12X4G12X4B12X4A12X4_2PACK16**
- **VK_FORMAT_R12X4G12X4B12X4A12X4_PACK16**

- **Extending VkFormatFeatureFlagBits:**
  - **VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT**
  - **VK_FORMAT_FEATURE_DISJOINT_BIT**
  - **VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT**
- **Extending** `VkImageAspectFlagBits`:
  - VK_IMAGE_ASPECT_PLANE_0_BIT
  - VK_IMAGE_ASPECT_PLANE_1_BIT
  - VK_IMAGE_ASPECT_PLANE_2_BIT

- **Extending** `VkImageCreateFlagBits`:
  - VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT
  - VK_IMAGE_CREATE_ALIAS_BIT
  - VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT
  - VK_IMAGE_CREATE_DISJOINT_BIT
  - VK_IMAGE_CREATE_EXTENDED_USAGE_BIT
  - VK_IMAGE_CREATE_PROTECTED_BIT
  - VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT

- **Extending** `VkImageLayout`:
  - VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_STENCIL_READ_ONLY_OPTIMAL
  - VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_STENCIL_ATTACHMENT_OPTIMAL

- **Extending** `VkMemoryHeapFlagBits`:
  - VK_MEMORY_HEAP_MULTI_INSTANCE_BIT

- **Extending** `VkMemoryPropertyFlagBits`:
  - VK_MEMORY_PROPERTY_PROTECTED_BIT

- **Extending** `VkObjectType`:
  - VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE
  - VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION

- **Extending** `VkPipelineCreateFlagBits`:
  - VK_PIPELINE_CREATE_DISPATCH_BASE_BIT
  - VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT

- **Extending** `VkQueueFlagBits`:
  - VK_QUEUE_PROTECTED_BIT

- **Extending** `VkResult`:
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO
  ◦ VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO
  ◦ VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO
  ◦ VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO
  ◦ VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO
  ◦ VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2
  ◦ VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT
  ◦ VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO
  ◦ VK_STRUCTURE_TYPE_DEVICE_QUEUE_INFO_2
  ◦ VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO
  ◦ VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES
  ◦ VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2
  ◦ VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2
  ◦ VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2
  ◦ VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO
  ◦ VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2
  ◦ VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO
  ◦ VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO
  ◦ VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO
  ◦ VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS
Appendix E: Layers & Extensions (Informative)

Extensions to the Vulkan API can be defined by authors, groups of authors, and the Khronos Vulkan Working Group. In order not to compromise the readability of the Vulkan Specification, the core Specification does not incorporate most extensions. The online Registry of extensions is available at URL

https://www.khronos.org/registry/vulkan/

and allows generating versions of the Specification incorporating different extensions.

Most of the content previously in this appendix does not specify use of specific Vulkan extensions and layers, but rather specifies the processes by which extensions and layers are created. As of version 1.0.21 of the Vulkan Specification, this content has been migrated to the Vulkan Documentation and Extensions document. Authors creating extensions and layers must follow the mandatory procedures in that document.

The remainder of this appendix documents a set of extensions chosen when this document was built. Versions of the Specification published in the Registry include:

- Core API + mandatory extensions required of all Vulkan implementations.
- Core API + all registered and published Khronos (KHR) extensions.
- Core API + all registered and published extensions.

Extensions are grouped as Khronos KHR, multivendor EXT, and then alphabetically by author ID. Within each group, extensions are listed in alphabetical order by their name.

Note

As of the initial Vulkan 1.1 public release, the KHX author ID is no longer used. All KHX extensions have been promoted to KHR status. Previously, this author ID was used to indicate that an extension was experimental, and is being considered for standardization in future KHR or core Vulkan API versions. We no longer use this mechanism for exposing experimental functionality.

Some vendors may use an alternate author ID ending in X for some of their extensions. The exact meaning of such an author ID is defined by each vendor, and may not be equivalent to KHX, but it is likely to indicate a lesser degree of interface stability than a non-X extension from the same vendor.

List of Current Extensions

- VK_KHR_8bit_storage
- VK_KHR_acceleration_structure
- VK_KHR_android_surface
• VK_KHR_buffer_device_address
• VK_KHR_copy_commands2
• VK_KHR_create_renderpass2
• VK_KHR_deferred_host_operations
• VK_KHR_depth_stencil_resolve
• VK_KHR_display
• VK_KHR_display_swapchain
• VK_KHR_draw_indirect_count
• VK_KHR_driver_properties
• VK_KHR_external_fence_fd
• VK_KHR_external_fence_win32
• VK_KHR_external_memory_fd
• VK_KHR_external_memory_win32
• VK_KHR_external_semaphore_fd
• VK_KHR_external_semaphore_win32
• VK_KHR_fragment_shading_rate
• VK_KHR_get_display_properties2
• VK_KHR_get_surface_capabilities2
• VK_KHR_image_format_list
• VK_KHR_imageless_framebuffer
• VK_KHR_incremental_present
• VK_KHR_performance_query
• VK_KHR_pipeline_executable_properties
• VK_KHR_pipeline_library
• VK_KHR_push_descriptor
• VK_KHR_ray_query
• VK_KHR_ray_tracing_pipeline
• VK_KHR_sampler_mirror_clamp_to_edge
• VK_KHR_separate_depthStencil_layouts
• VK_KHR_shader_atomic_int64
• VK_KHR_shader_clock
• VK_KHR_shader_float16_int8
• VK_KHR_shader_float_controls
• VK_KHR_shader_non_semantic_info
• VK_KHR_shader_subgroup_extended_types
• VK_KHR_shader_terminate_invocation
• VK_KHR_shared_presentable_image
• VK_KHR_spirv_1_4
• VK_KHR_surface
• VK_KHR_surface_protected_capabilities
• VK_KHR_swapchain
• VK_KHR_swapchain_mutable_format
• VK_KHR_synchronization2
• VK_KHR_timeline_semaphore
• VK_KHR_uniform_buffer_standard_layout
• VK_KHR_vulkan_memory_model
• VK_KHR_wayland_surface
• VK_KHR_win32_keyed_mutex
• VK_KHR_win32_surface
• VK_KHR_workgroup_memory_explicit_layout
• VK_KHR xcb_surface
• VK_KHR_xlib_surface
• VK_KHR_zero_initialize_workgroup_memory
• VK_EXT_4444_formats
• VK_EXT_acquire_xlib_display
• VK_EXT_astc_decode_mode
• VK_EXT_blend_operation_advanced
• VK_EXT_calibrated_timestamps
• VK_EXT_color_write_enable
• VK_EXT_conditional_rendering
• VK_EXT_conservative_rasterization
• VK_EXT_custom_border_color
• VK_EXT_debug_utils
• VK_EXT_depth_clip_enable
• VK_EXT_depth_range_unrestricted
• VK_EXT_descriptor_indexing
• VK_EXT_device_memory_report
• VK_EXT_direct_mode_display
• VK_EXT_directfb_surface
• VK_EXT_discard_rectangles
• VK_EXT_display_control
• VK_EXT_display_surface_counter
• VK_EXT_extended_dynamic_state
• VK_EXT_extended_dynamic_state2
• VK_EXT_external_memory_dma_buf
• VK_EXT_external_memory_host
• VK_EXT_filter_cubic
• VK_EXT_fragment_density_map
• VK_EXT_fragment_density_map2
• VK_EXT_fragment_shader_interlock
• VK_EXT_full_screen_exclusive
• VK_EXT_global_priority
• VK_EXT_hdr_metadata
• VK_EXT_headless_surface
• VK_EXT_host_query_reset
• VK_EXT_image_drm_format_modifier
• VK_EXT_image_robustness
• VK_EXT_index_type_uint8
• VK_EXT_inline_uniform_block
• VK_EXT_line_rasterization
• VK_EXT_memory_budget
• VK_EXT_memory_priority
• VK_EXT_metal_surface
• VK_EXT_pci_bus_info
• VK_EXT_pipeline_creation_cache_control
• VK_EXT_pipeline_creation_feedback
• VK_EXT_post_depth_coverage
• VK_EXT_private_data
• VK_EXT_provoking_vertex
• VK_EXT_queue_family_foreign
• VK_EXT_robustness2
• VK_EXT_sample_locations
• VK_EXT_sampler_filter_minmax
• VK_EXT_scalar_block_layout
• VK_EXT_separate_stencil_usage
• VK_EXT_shader_atomic_float
• VK_EXT_shader_demote_to_helper_invocation
• VK_EXT_shader_image_atomic_int64
• VK_EXT_shader_stencil_export
• VK_EXT_shader_subgroup_ballot
• VK_EXT_shader_viewport_index_layer
• VK_EXT_subgroup_size_control
• VK_EXT_swapchain_colorsapce
• VK_EXT_texel_buffer_alignment
• VK_EXT_texture_compression_astc_hdr
• VK_EXT_tooling_info
• VK_EXT_transform_feedback
• VK_EXT_validation_cache
• VK_EXT_validation_features
• VK_EXT_vertex_attribute_divisor
• VK_EXT_vertex_input_dynamic_state
• VK_EXT_ycbcr_2plane_444_formats
• VK_EXT_ycbcr_image_arrays
• VK_AMD_buffer_marker
• VK_AMD_device_coherent_memory
• VK_AMD_display_native_hdr
• VK_AMD_gcn_shader
• VK_AMD_memory_overallocation_behavior
• VK_AMD_mixed_attachment_samples
• VK_AMD_pipeline_compiler_control
• VK_AMD_rasterization_order
• VK_AMD_shader_ballot
• VK_AMD_shader_core_properties
• VK_AMD_shader_core_properties2
• VK_AMD_shader_explicit_vertex_parameter
• VK_AMD_shader_fragment_mask
• VK_AMD_shader_image_load_store_lod
• VK_AMD_shader_info
• VK_AMD_shader_trinary_minmax
• VK_AMD_texture_gather_bias_lod
• VK_ANDROID_external_memory_android_hardware_buffer
• VK_FUCHSIA_external_memory
• VK_FUCHSIA_external_semaphore
• VK_FUCHSIA_imagepipe_surface
• VK_GGP_frame_token
• VK_GGP_stream_descriptor_surface
• VK_GOOGLE_decorate_string
• VK_GOOGLE_display_timing
• VK_GOOGLE_hls1_functionality1
• VK_GOOGLE_user_type
• VK_IMG_filter_cubic
• VK_IMG_format_pvrtc
• VK_INTEL_performance_query
• VK_INTEL_shader_integer_functions2
• VK_NN_vi_surface
• VK_NV_acquire_winrt_display
• VK_NV_clip_space_w_scaling
• VK_NV_compute_shader_derivatives
• VK_NV_cooperative_matrix
• VK_NV_corner_sampled_image
• VK_NV_coverage_reduction_mode
• VK_NV_dedicated_allocation_image_aliasing
• VK_NV_device_diagnostic_checkpoints
• VK_NV_device_diagnostics_config
• VK_NV_device_generated_commands
• VK_NV_fill_rectangle
• VK_NV_fragment_coverage_to_color
• VK_NV_fragment_shader_barycentric
• VK_NV_fragment_shading_rate Enums
• VK_NV_framebuffer_mixed_samples
• VK_NV_geometry_shader_passthrough
• VK_NV_inherited_viewport_scissor
• VK_NV_mesh_shader
• VK_NV_ray_tracing
• VK_NV_representative_fragment_test
• VK_NV_sample_mask_override_coverage
• VK_NV_scissor_exclusive
• VK_NV_shader_image_footprint
• VK_NV_shader_sm_builtins
• VK_NV_shader_subgroup_partitioned
• VK_NV_shading_rate_image
• VK_NV_viewport_array2
• VK_NV_viewport_swizzle
• VK_NVX_image_view_handle
• VK_NVX_multiview_per_view_attributes
• VK_QCOM_render_pass_shader_resolve
• VK_QCOM_render_pass_store_ops
• VK_QCOM_render_pass_transform
• VK_QCOM_rotated_copy_commands
• VK_QNX_screen_surface
• VK_VALVE_mutable_descriptor_type
**VK_KHR_8bit_storage**

**Name String**

*VK_KHR_8bit_storage*

**Extension Type**

Device extension

**Registered Extension Number**

178

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires *VK_KHR_get_physical_device_properties2*
- Requires *VK_KHR_storage_buffer_storage_class*

**Deprecation state**

- *Promoted* to Vulkan 1.2

**Contact**

- Alexander Galazin [alegal-arm](mailto:alegal-arm)

**Other Extension Metadata**

**Last Modified Date**

2018-02-05

**Interactions and External Dependencies**

- Promoted to Vulkan 1.2 Core
- This extension requires *SPV_KHR_8bit_storage*
- This extension provides API support for *GL_EXT_shader_16bit_storage*

**IP Status**

No known IP claims.

**Contributors**

- Alexander Galazin, Arm

**Description**

The *VK_KHR_8bit_storage* extension allows use of 8-bit types in uniform and storage buffers, and push constant blocks. This extension introduces several new optional features which map to SPIR-V capabilities and allow access to 8-bit data in *Block*-decorated objects in the *Uniform* and the *StorageBuffer* storage classes, and objects in the *PushConstant* storage class.
The `StorageBuffer8BitAccess` capability must be supported by all implementations of this extension. The other capabilities are optional.

**Promotion to Vulkan 1.2**

Functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the `StorageBuffer8BitAccess` capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDevice8BitStorageFeaturesKHR`

**New Enum Constants**

- `VK_KHR_8BIT_STORAGE_EXTENSION_NAME`
- `VK_KHR_8BIT_STORAGE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_8BIT_STORAGE_FEATURES_KHR`

**New SPIR-V Capabilities**

- `StorageBuffer8BitAccess`
- `UniformAndStorageBuffer8BitAccess`
- `StoragePushConstant8`

**Version History**

- Revision 1, 2018-02-05 (Alexander Galazin)
  - Initial draft

**VK_KHR_acceleration_structure**

**Name String**

`VK_KHR_acceleration_structure`

**Extension Type**

Device extension

**Registered Extension Number**

151

**Revision**

11
Extension and Version Dependencies

- Requires Vulkan 1.1
- Requires VK_EXT_descriptor_indexing
- Requires VK_KHR_buffer_device_address
- Requires VK_KHR_deferred_host_operations

Contact

- Daniel Koch @dgkoch

Other Extension Metadata

Last Modified Date
2020-11-12

Contributors

- Matthäus Chajdas, AMD
- Greg Grebe, AMD
- Nicolai Hähnle, AMD
- Tobias Hector, AMD
- Dave Oldcorn, AMD
- Skyler Saleh, AMD
- Mathieu Robart, Arm
- Marius Bjorge, Arm
- Tom Olson, Arm
- Sebastian Tafuri, EA
- Henrik Rydgard, Embark
- Juan Cañada, Epic Games
- Patrick Kelly, Epic Games
- Yuriy O'Donnell, Epic Games
- Michael Doggett, Facebook/Oculus
- Ricardo Garcia, Igalia
- Andrew Garrard, Imagination
- Don Scorgie, Imagination
- Dae Kim, Imagination
- Joshua Barczak, Intel
- Slawek Grajewski, Intel
- Jeff Bolz, NVIDIA
- Pascal Gautron, NVIDIA
Description

In order to be efficient, rendering techniques such as ray tracing need a quick way to identify which primitives may be intersected by a ray traversing the geometries. Acceleration structures are the most common way to represent the geometry spatially sorted, in order to quickly identify such potential intersections.

This extension adds new functionalities:

- Acceleration structure objects and build commands
- Structures to describe geometry inputs to acceleration structure builds
- Acceleration structure copy commands

New Object Types

- VkAccelerationStructureKHR

New Commands

- vkBuildAccelerationStructuresKHR
- vkCmdBuildAccelerationStructuresIndirectKHR
- `vkCmdBuildAccelerationStructuresKHR`  
- `vkCmdCopyAccelerationStructureKHR`  
- `vkCmdCopyAccelerationStructureToMemoryKHR`  
- `vkCmdCopyMemoryToAccelerationStructureKHR`  
- `vkCmdWriteAccelerationStructuresPropertiesKHR`  
- `vkCopyAccelerationStructureKHR`  
- `vkCopyAccelerationStructureToMemoryKHR`  
- `vkCopyMemoryToAccelerationStructureKHR`  
- `vkCreateAccelerationStructureKHR`  
- `vkDestroyAccelerationStructureKHR`  
- `vkGetAccelerationStructureBuildSizesKHR`  
- `vkGetAccelerationStructureDeviceAddressKHR`  
- `vkGetDeviceAccelerationStructureCompatibilityKHR`  
- `vkWriteAccelerationStructuresPropertiesKHR`  

**New Structures**

- `VkAabbPositionsKHR`  
- `VkAccelerationStructureBuildGeometryInfoKHR`  
- `VkAccelerationStructureBuildRangeInfoKHR`  
- `VkAccelerationStructureBuildSizesInfoKHR`  
- `VkAccelerationStructureCreateInfoKHR`  
- `VkAccelerationStructureDeviceAddressInfoKHR`  
- `VkAccelerationStructureGeometryAabbsDataKHR`  
- `VkAccelerationStructureGeometryInstancesDataKHR`  
- `VkAccelerationStructureGeometryKHR`  
- `VkAccelerationStructureGeometryTrianglesDataKHR`  
- `VkAccelerationStructureInstanceKHR`  
- `VkAccelerationStructureVersionInfoKHR`  
- `VkCopyAccelerationStructureInfoKHR`  
- `VkCopyAccelerationStructureToMemoryInfoKHR`  
- `VkCopyMemoryToAccelerationStructureInfoKHR`  
- `VkTransformMatrixKHR`  

**Extending** `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:

- `VkPhysicalDeviceAccelerationStructureFeaturesKHR`  

**Extending** `VkPhysicalDeviceProperties2`:
• VkPhysicalDeviceAccelerationStructurePropertiesKHR

Extending VkWriteDescriptorSet:
• VkWriteDescriptorSetAccelerationStructureKHR

New Unions
• VkAccelerationStructureGeometryDataKHR
• VkDeviceOrHostAddressConstKHR
• VkDeviceOrHostAddressKHR

New Enums
• VkAccelerationStructureBuildTypeKHR
• VkAccelerationStructureCompatibilityKHR
• VkAccelerationStructureCreateFlagBitsKHR
• VkAccelerationStructureTypeKHR
• VkBuildAccelerationStructureFlagBitsKHR
• VkCopyAccelerationStructureModeKHR
• VkGeometryFlagBitsKHR
• VkGeometryInstanceFlagBitsKHR
• VkGeometryTypeKHR

New Bitmasks
• VkAccelerationStructureCreateFlagsKHR
• VkBuildAccelerationStructureFlagsKHR
• VkGeometryFlagsKHR
• VkGeometryInstanceFlagsKHR

New Enum Constants
• VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME
• VK_KHR_ACCELERATION_STRUCTURE_SPEC_VERSION

Extending VkAccessFlagBits:
• VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_KHR
• VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_KHR

Extending VkBufferUsageFlagBits:
• VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR
• Extending `VkDebugReportObjectTypeEXT`:
  ◦ `VK_DEBUG_REPORT_OBJECT_TYPE_ACCELERATION_STRUCTURE_KHR_EXT`  

• Extending `VkDescriptorType`:
  ◦ `VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR`  

• Extending `VkFormatFeatureFlagBits`:
  ◦ `VK_FORMAT_FEATURE_ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR`  

• Extending `VkIndexType`:
  ◦ `VK_INDEX_TYPE_NONE_KHR`  

• Extending `VkObjectType`:
  ◦ `VK_OBJECT_TYPE_ACCELERATION_STRUCTURE_KHR`  

• Extending `VkPipelineStageFlagBits`:
  ◦ `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`  

• Extending `VkQueryType`:
  ◦ `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_COMPACTED_SIZE_KHR`  
  ◦ `VK_QUERY_TYPE_ACCELERATION_STRUCTURE_SERIALIZATION_SIZE_KHR`  

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_DEVICE_ADDRESS_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_AABBS_DATA_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR`  
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_VERSION_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_COPY_ACCELERATION_STRUCTURE_TO_MEMORY_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_COPY_MEMORY_TO_ACCELERATION_STRUCTURE_INFO_KHR`  
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ACCELERATION_STRUCTURE_FEATURES_KHR`  
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL DEVICE_ACCELERATION_STRUCTURE_PROPERTIES_KHR`  
  ◦ `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR SET_ACCELERATION_STRUCTURE_KHR`  

**Issues**

(1) How does this extension differ from VK_NV_ray_tracing?
DISCUSSION:

The following is a summary of the main functional differences between VK_KHR_acceleration_structure and VK_NV_ray_tracing:

• added acceleration structure serialization / deserialization
  (VK_COPY_ACCELERATION_STRUCTURE_MODE_SERIALIZE_KHR,
  VK_COPY_ACCELERATION_STRUCTURE_MODE_DESERIALIZE_KHR,
  vkCmdCopyAccelerationStructureToMemoryKHR,
  vkCmdCopyMemoryToAccelerationStructureKHR)

• document inactive primitives and instances

• added VkPhysicalDeviceAccelerationStructureFeaturesKHR structure

• added indirect and batched acceleration structure builds
  (vkCmdBuildAccelerationStructuresIndirectKHR)

• added host acceleration structure commands

• reworked geometry structures so they could be better shared between device, host, and indirect builds

• explicitly made VkAccelerationStructureKHR use device addresses

• added acceleration structure compatibility check function
  (vkGetDeviceAccelerationStructureCompatibilityKHR)

• add parameter for requesting memory requirements for host and/or device build

• added format feature for acceleration structure build vertex formats
  (VK_FORMAT_FEATURE_ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR)

(2) Can you give a more detailed comparison of differences and similarities between VK_NV_ray_tracing and VK_KHR_acceleration_structure?

DISCUSSION:

The following is a more detailed comparison of which commands, structures, and enums are aliased, changed, or removed.

• Aliased functionality — enums, structures, and commands that are considered equivalent:
  ◦ VkGeometryTypeNV ↔ VkGeometryTypeKHR
  ◦ VkAccelerationStructureTypeNV ↔ VkAccelerationStructureTypeKHR
  ◦ VkCopyAccelerationStructureModeNV ↔ VkCopyAccelerationStructureModeKHR
  ◦ VkGeometryFlagsNV ↔ VkGeometryFlagsKHR
  ◦ VkGeometryFlagBitsNV ↔ VkGeometryFlagBitsKHR
  ◦ VkGeometryInstanceFlagsNV ↔ VkGeometryInstanceFlagsKHR
  ◦ VkGeometryInstanceFlagBitsNV ↔ VkGeometryInstanceFlagBitsKHR
  ◦ VkBuildAccelerationStructureFlagsNV ↔ VkBuildAccelerationStructureFlagsKHR
  ◦ VkBuildAccelerationStructureFlagBitsNV ↔ VkBuildAccelerationStructureFlagBitsKHR
- \textbf{VkTransformMatrixNV} $\leftrightarrow$ \textbf{VkTransformMatrixKHR} (added to VK_NV_ray_tracing for descriptive purposes)
- \textbf{VkAabbPositionsNV} $\leftrightarrow$ \textbf{VkAabbPositionsKHR} (added to VK_NV_ray_tracing for descriptive purposes)
- \textbf{VkAccelerationStructureInstanceNV} $\leftrightarrow$ \textbf{VkAccelerationStructureInstanceKHR} (added to VK_NV_ray_tracing for descriptive purposes)

- Changed enums, structures, and commands:
  - renamed \textbf{VK_GEOMETRY_INSTANCE_TRIANGLE_CULL_DISABLE_BIT_NV} $\rightarrow$ \textbf{VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR} in \textbf{VkGeometryInstanceFlagBitsKHR}
  - \textbf{VkGeometryTrianglesNV} $\rightarrow$ \textbf{VkAccelerationStructureGeometryTrianglesDataKHR} (device or host address instead of buffer+offset)
  - \textbf{VkGeometryAABBNV} $\rightarrow$ \textbf{VkAccelerationStructureGeometryAabbsDataKHR} (device or host address instead of buffer+offset)
  - \textbf{VkGeometryDataNV} $\rightarrow$ \textbf{VkAccelerationStructureGeometryDataKHR} (union of triangle/aabbs/instances)
  - \textbf{VkGeometryNV} $\rightarrow$ \textbf{VkAccelerationStructureGeometryKHR} (changed type of geometry)
  - \textbf{VkAccelerationStructureCreateInfoNV} $\rightarrow$ \textbf{VkAccelerationStructureCreateInfoKHR} (reshuffle geometry layout/info)
  - \textbf{VkPhysicalDeviceRayTracingPropertiesNV} $\rightarrow$ \textbf{VkPhysicalDeviceAccelerationStructurePropertiesKHR} (for acceleration structure properties, renamed \textbf{maxTriangleCount} to \textbf{maxPrimitiveCount}, added per stage and update after bind limits) and \textbf{VkPhysicalDeviceRayTracingPipelinePropertiesKHR} (for ray tracing pipeline properties)
  - \textbf{VkAccelerationStructureMemoryRequirementsInfoNV} (deleted - replaced by allocating on top of \textbf{VkBuffer})
  - \textbf{VkWriteDescriptorSetAccelerationStructureNV} $\rightarrow$ \textbf{VkWriteDescriptorSetAccelerationStructureKHR} (different acceleration structure type)
  - \textbf{vkCreateAccelerationStructureNV} $\rightarrow$ \textbf{vkCreateAccelerationStructureKHR} (device address, different geometry layout/info)
  - \textbf{vkGetAccelerationStructureMemoryRequirementsNV} (deleted - replaced by allocating on top of \textbf{VkBuffer})
  - \textbf{vkCmdBuildAccelerationStructureNV} $\rightarrow$ \textbf{vkCmdBuildAccelerationStructuresKHR} (params moved to structs, layout differences)
  - \textbf{vkCmdCopyAccelerationStructureNV} $\rightarrow$ \textbf{vkCmdCopyAccelerationStructureKHR} (params to struct, extendable)
  - \textbf{vkGetAccelerationStructureHandleNV} $\rightarrow$ \textbf{vkGetAccelerationStructureDeviceAddressKHR} (device address instead of handle)
  - \textbf{VkAccelerationStructureMemoryRequirementsTypeNV} $\rightarrow$ size queries for scratch space moved to \textbf{vkGetAccelerationStructureBuildSizesKHR}

2510 | Appendix E: Layers & Extensions (Informative)
• `vkDestroyAccelerationStructureNV` → `vkDestroyAccelerationStructureKHR`  (different acceleration structure types)

• `vkCmdWriteAccelerationStructuresPropertiesNV` → `vkCmdWriteAccelerationStructuresPropertiesKHR`  (different acceleration structure types)

• **Added enums, structures and commands:**
  - `VK_GEOMETRY_TYPE_INSTANCES_KHR` to `VkGeometryTypeKHR` enum
  - `VK_COPY_ACCELERATION_STRUCTURE_MODE_SERIALIZE_KHR, VK_COPY_ACCELERATION_STRUCTURE_MODE_DESERIALIZE_KHR` to `VkCopyAccelerationStructureModeKHR` enum
  - `VkPhysicalDeviceAccelerationStructureFeaturesKHR` structure
  - `VkAccelerationStructureBuildTypeKHR` enum
  - `VkBuildAccelerationStructureModeKHR` enum
  - `VkDeviceOrHostAddressKHR` and `VkDeviceOrHostAddressConstKHR` unions
  - `VkAccelerationStructureBuildRangeInfoKHR` struct
  - `VkAccelerationStructureGeometryInstancesDataKHR` struct
  - `VkAccelerationStructureDeviceAddressInfoKHR` struct
  - `VkAccelerationStructureVersionInfoKHR` struct
  - `VkStridedDeviceAddressRegionKHR` struct
  - `VkCopyAccelerationStructureToMemoryInfoKHR` struct
  - `VkCopyMemoryToAccelerationStructureInfoKHR` struct
  - `VkCopyAccelerationStructureInfoKHR` struct
  - `vkBuildAccelerationStructuresKHR` command (host build)
  - `vkCopyAccelerationStructureKHR` command (host copy)
  - `vkCopyAccelerationStructureToMemoryKHR` (host serialize)
  - `vkCopyMemoryToAccelerationStructureKHR` (device deserialize)
  - `vkWriteAccelerationStructuresPropertiesKHR` (host deserialize)
  - `vkCmdCopyAccelerationStructureToMemoryKHR` (device serialize)
  - `vkCmdCopyMemoryToAccelerationStructureKHR` (device deserialize)
  - `vkGetDeviceAccelerationStructureCompatibilityKHR` (serialization)

(3) What are the changes between the public provisional (VK_KHR_ray_tracing v8) release and the internal provisional (VK_KHR_ray_tracing v9) release?

• added `geometryFlags` to `VkAccelerationStructureCreateGeometryTypeInfoKHR` (later reworked to obsolete this)
• added `minAccelerationStructureScratchOffsetAlignment` property to `VkPhysicalDeviceRayTracingPropertiesKHR`
• fix naming and return enum from `vkGetDeviceAccelerationStructureCompatibilityKHR`
• renamed `VkAccelerationStructureVersionKHR` to `VkAccelerationStructureVersionInfoKHR`

• renamed `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_VERSION_KHR` to `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_VERSION_INFO_KHR`

• removed `VK_ERROR_INCOMPATIBLE_VERSION_KHR`

• added `VkAccelerationStructureCompatibilityKHR` enum

• remove return value from `vkGetDeviceAccelerationStructureCompatibilityKHR` and added return enum parameter

• Require Vulkan 1.1

• added creation time capture and replay flags

• added `VkAccelerationStructureCreateFlagBitsKHR` and `VkAccelerationStructureCreateFlagsKHR`

• renamed the `flags` member of `VkAccelerationStructureCreateInfoKHR` to `buildFlags` (later removed) and added the `createFlags` member

• change `vkCmdBuildAccelerationStructuresIndirectKHR` to use buffer device address for indirect parameter

• make `VK_KHR_deferred_host_operations` an interaction instead of a required extension (later went back on this)

• renamed `VkAccelerationStructureBuildOffsetInfoKHR` to `VkAccelerationStructureBuildRangeInfoKHR`

• renamed the `ppOffsetInfos` parameter of `vkCmdBuildAccelerationStructuresKHR` to `ppBuildRangeInfos`

• Re-unify geometry description between build and create

• remove `VkAccelerationStructureCreateGeometryTypeInfoKHR` and `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_GEOMETRY_TYPE_INFO_KHR`

• added `VkAccelerationStructureCreateSizeInfoKHR` structure (later removed)

• change type of the `pGeometryInfos` member of `VkAccelerationStructureCreateInfoKHR` from `VkAccelerationStructureCreateGeometryTypeInfoKHR` to `VkAccelerationStructureGeometryKHR` (later removed)

• added `pCreateSizeInfos` member to `VkAccelerationStructureCreateInfoKHR` (later removed)

• Fix ppGeometries ambiguity, add pGeometries

• remove `geometryArrayOfPointers` member of `VkAccelerationStructureBuildGeometryInfoKHR`

• disambiguate two meanings of `ppGeometries` by explicitly adding `pGeometries` to the `VkAccelerationStructureBuildGeometryInfoKHR` structure and require one of them be `NULL`

• added `nullDescriptor` support for acceleration structures

• changed the `update` member of `VkAccelerationStructureBuildGeometryInfoKHR` from a bool to the `mode` `VkBuildAccelerationStructureModeKHR` enum which allows future extensibility in update types

• Clarify deferred host ops for pipeline creation
- **VkDeferredOperationKHR** is now a top-level parameter for
  vkBuildAccelerationStructuresKHR, vkCreateRayTracingPipelinesKHR,
  vkCopyAccelerationStructureToMemoryKHR, vkCopyAccelerationStructureKHR, and
  vkCopyMemoryToAccelerationStructureKHR

- removed **VkDeferredOperationInfoKHR** structure

- change deferred host creation/return parameter behavior such that the implementation can
  modify such parameters until the deferred host operation completes

- **VK_KHR_deferred_host_operations** is required again

  - Change acceleration structure build to always be sized
    - de-alias **VkAccelerationStructureMemoryRequirementsTypeNV** and
      **VkAccelerationStructureMemoryRequirementsTypeKHR** and remove
      **VkAccelerationStructureMemoryRequirementsTypeKHR**
    - add **vkGetAccelerationStructureBuildSizesKHR** command and
      **VkAccelerationStructureBuildSizesInfoKHR** structure and
      **VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR** enum to query sizes for
      acceleration structures and scratch storage
    - move size queries for scratch space to **vkGetAccelerationStructureBuildSizesKHR**
    - remove **compactedSize**, **buildFlags**, **maxGeometryCount**, **pGeometryInfos**, **pCreateSizeInfos**
      members of **VkAccelerationStructureCreateInfoKHR** and add the **size** member
    - add **maxVertex** member to **VkAccelerationStructureGeometryTrianglesDataKHR** structure
    - remove **VkAccelerationStructureCreateSizeInfoKHR** structure

(4) What are the changes between the internal provisional (VK_KHR_ray_tracing v9) release and the
final (VK_KHR_acceleration_structure v11) release?

  - refactor **VK_KHR_ray_tracing** into 3 extensions, enabling implementation flexibility and
decoupling ray query support from ray pipelines:
    - **VK_KHR_acceleration_structure** (for acceleration structure operations)
    - **VK_KHR_ray_tracing_pipeline** (for ray tracing pipeline and shader stages)
    - **VK_KHR_ray_query** (for ray queries in existing shader stages)

  - clarify buffer usage flags for ray tracing
    - **VK_BUFFER_USAGE_RAY_TRACING_BIT_NV** is left alone in **VK_NV_ray_tracing** (required on **scratch**
      and **instanceData**)
    - **VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR** is added as an alias of
      **VK_BUFFER_USAGE_RAY_TRACING_BIT_NV** in **VK_KHR_ray_tracing_pipeline** and is required on
      shader binding table buffers
    - **VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR** is added in
      **VK_KHR_acceleration_structure** for all vertex, index, transform, aabb, and instance buffer
      data referenced by device build commands
  - **VK_BUFFER_USAGE_STORAGE_BUFFER_BIT** is used for **scratchData**

  - add max primitive counts (ppMaxPrimitiveCounts) to
vkCmdBuildAccelerationStructuresIndirectKHR

- Allocate acceleration structures from **VkBuffers** and add a mode to constrain the device address
  - de-alias **VkBindAccelerationStructureMemoryInfoNV**, **VkBindAccelerationStructureMemoryNV**, **VkBindAccelerationStructureMemoryInfoKHR**, **VkAccelerationStructureMemoryRequirementsInfoKHR**, and **vkGetAccelerationStructureMemoryRequirementsKHR**
  - acceleration structures now take a **VkBuffer** and offset at creation time for memory placement
  - add a new **VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_STORAGE_BIT_KHR** buffer usage for such buffers
  - add a new **VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR** acceleration structure type for layering
- move **VK_GEOMETRY_TYPE_INSTANCES_KHR** to main enum instead of being added via extension
- make build commands more consistent - all now build multiple acceleration structures and are named plural ly (**vkCmdBuildAccelerationStructuresIndirectKHR**, **vkCmdBuildAccelerationStructuresKHR**, **vkBuildAccelerationStructuresKHR**)
- add interactions with **VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT** for acceleration structures, including a new feature (**descriptorBindingAccelerationStructureUpdateAfterBind**), and 3 new properties (**maxPerStageDescriptorAccelerationStructures**, **maxPerStageDescriptorUpdateAfterBindAccelerationStructures**, **maxDescriptorSetUpdateAfterBindAccelerationStructures**)
- extension is no longer provisional
- define synchronization requirements for builds, traces, and copies
- define synchronization requirements for AS build inputs and indirect build buffer

(5) What is **VK_ACCELERATION_STRUCTURE_TYPE_GENERIC_KHR** for?

**RESOLVED:** It is primarily intended for API layering. In DXR, the acceleration structure is basically just a buffer in a special layout, and you don’t know at creation time whether it will be used as a top or bottom level acceleration structure. We thus added a generic acceleration structure type whose type is unknown at creation time, but is specified at build time instead. Applications which are written directly for Vulkan should not use it.

**Version History**

- Revision 1, 2019-12-05 (Members of the Vulkan Ray Tracing TSG)
  - Internal revisions (forked from VK_NV_ray_tracing)
- Revision 2, 2019-12-20 (Daniel Koch, Eric Werness)
  - Add const version of DeviceOrHostAddress (!3515)
  - Add VU to clarify that only handles in the current pipeline are valid (!3518)
• Restore some missing VUs and add in-place update language (#1902, !3522)
• rename VkAccelerationStructureInstanceKHR member from accelerationStructure to accelerationStructureReference to better match its type (!3523)
• Allow VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS for pipeline creation if shader group handles cannot be reused (!3523)
• update documentation for the VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS error code and add missing documentation for new return codes from VK_KHR_deferred_host_operations (!3523)
• list new query types for VK_KHR_ray_tracing (!3523)
• Fix VU statements for VkAccelerationStructureGeometryKHR referring to correct union members and update to use more current wording (!3523)

• Revision 3, 2020-01-10 (Daniel Koch, Jon Leech, Christoph Kubisch)
  • Fix 'instance of' and 'that/which contains/defines' markup issues (!3528)
  • factor out VK_KHR_pipeline_library as stand-alone extension (!3540)
  • Resolve Vulkan-hpp issues (!3543)
  • add missing require for VkGeometryInstanceFlagsKHR
  • de-alias VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV since the KHR structure is no longer equivalent
  • add len to pDataSize attribute for vkWriteAccelerationStructuresPropertiesKHR

• Revision 4, 2020-01-23 (Daniel Koch, Eric Werness)
  • Improve vkWriteAccelerationStructuresPropertiesKHR, add return value and VUs (#1947)
  • Clarify language to allow multiple raygen shaders (#1959)
  • Various editorial feedback (!3556)
  • Add language to help deal with looped self-intersecting fans (#1901)
  • Change vkCmdTraceRays(Indirect)KHR args to pointers (!3559)
  • Add scratch address validation language (#1941, !3551)
  • Fix definition and add hierarchy information for shader call scope (#1977, !3571)

• Revision 5, 2020-02-04 (Eric Werness, Jeff Bolz, Daniel Koch)
  • remove vestigial accelerationStructureUUID (!3582)
  • update definition of repack instructions and improve memory model interactions (#1910, #1913, !3584)
  • Fix wrong sType for VkPhysicalDeviceRayTracingFeaturesKHR (#1988)
  • Use provisional SPIR-V capabilities (#1987)
  • require rayTraversalPrimitiveCulling if rayQuery is supported (#1927)
  • Miss shaders do not have object parameters (!3592)
  • Fix missing required types in XML (!3592)
  • clarify matching conditions for update (!3592)
- add goal that host and device builds be similar (!3592)
- clarify that \texttt{maxPrimitiveCount} limit should apply to triangles and AABBs (!3592)
- Require alignment for instance arrayOfPointers (!3592)
- Zero is a valid value for instance flags (!3592)
- Add some alignment VUs that got lost in refactoring (!3592)
- Recommend TMin epsilon rather than culling (!3592)
- Get angle from dot product not cross product (!3592)
- Clarify that AH can access the payload and attributes (!3592)
- Match DXR behavior for inactive primitive definition (!3592)
- Use a more generic term than degenerate for inactive to avoid confusion (!3592)

**Revision 6, 2020-02-20 (Daniel Koch)**

- fix some dangling NV references (#1996)
- rename \texttt{VkCmdTraceRaysIndirectCommandKHR} to \texttt{VkTraceRaysIndirectCommandKHR} (!3607)
- update contributor list (!3611)
- use \texttt{uint64_t} instead of \texttt{VkAccelerationStructureReferenceKHR} in \texttt{VkAccelerationStructureInstanceKHR} (#2004)

**Revision 7, 2020-02-28 (Tobias Hector)**

- remove HitTKHR SPIR-V builtin (spirv/spirv-extensions#7)

**Revision 8, 2020-03-06 (Tobias Hector, Dae Kim, Daniel Koch, Jeff Bolz, Eric Werness)**

- explicitly state that Tmax is updated when new closest intersection is accepted (#2020,!3536)
- Made references to min and max t values consistent (!3644)
- finish enumerating differences relative to \texttt{VK_NV_ray_tracing} in issues (1) and (2) (#1974,!3642)
- fix formatting in some math equations (!3642)
- Restrict the Hit Kind operand of \texttt{OpReportIntersectionKHR} to 7-bits (spirv/spirv-extensions#8,!3646)
- Say ray tracing 'should' be watertight (#2008,!3631)
- Clarify memory requirements for ray tracing buffers (#2005,!3649)
- Add callable size limits (#1997,!3652)

**Revision 9, 2020-04-15 (Eric Werness, Daniel Koch, Tobias Hector, Joshua Barczak)**

- Add geometry flags to acceleration structure creation (!3672)
- add build scratch memory alignment (minAccelerationStructureScratchOffsetAlignment) (#2065,!3725)
- fix naming and return enum from \texttt{vkGetDeviceAccelerationStructureCompatibilityKHR} (#2051,!3726)
• require SPIR-V 1.4 (#2096,13777)
• added creation time capture/replay flags (#2104,13774)
• require Vulkan 1.1 (#2133,13806)
• use device addresses instead of VkBuffers for ray tracing commands (#2074,13815)
• add interactions with Vulkan 1.2 and VK_KHR_vulkan_memory_model (#2133,13830)
• make VK_KHR_pipeline_library an interaction instead of required (#2045,#2108,13830)
• make VK_KHR_deferred_host_operations an interaction instead of required (#2045,13830)
• removed maxCallableSize and added explicit stack size management for ray pipelines (#1997,13717,13772,13844)
• improved documentation for VkAccelerationStructureVersionInfoKHR (#2135,3835)
• renamed VkAccelerationStructureBuildOffsetInfoKHR to VkAccelerationStructureBuildRangeInfoKHR (#2058,13754)
• Re-unify geometry description between build and create (#3754)
• Fix ppGeometries ambiguity, add pGeometries (#2032,13811)
• add interactions with VK_EXT_robustness2 and allow nullDescriptor support for acceleration structures (#1920,13848)
• added future extensibility for AS updates (#2114,13849)
• Fix VU for dispatchrays and add a limit on the size of the full grid (#2160,13851)
• Add shaderGroupHandleAlignment property (#2180,13875)
• Clarify deferred host ops for pipeline creation (#2067,13813)
• Change acceleration structure build to always be sized (#2131,#2197,#2198,13854,13883,13880)

• Revision 10, 2020-07-03 (Mathieu Robart, Daniel Koch, Eric Werness, Tobias Hector)

• Decomposition of the specification, from VK_KHR_ray_tracing to VK_KHR_acceleration_structure (#1918,13912)
• clarify buffer usage flags for ray tracing (#2181,13939)
• add max primitive counts to build indirect command (#2233,13944)
• Allocate acceleration structures from VkBuffers and add a mode to constrain the device address (#2131,13936)
• Move VK_GEOMETRY_TYPE_INSTANCES_KHR to main enum (#2243,13952)
• make build commands more consistent (#2247,13958)
• add interactions with UPDATE_AFTER_BIND (#2128,13986)
• correct and expand build command VUs (!4020)
• fix copy command VUs (!4018)
• added various alignment requirements (#2229,13943)
• fix valid usage for arrays of geometryCount items (#2198,14010)
- define what is allowed to change on RTAS updates and relevant VUs (#2177,13961)

Revision 11, 2020-11-12 (Eric Werness, Josh Barczak, Daniel Koch, Tobias Hector)
- de-alias NV and KHR acceleration structure types and associated commands (#2271,14035)
- specify alignment for host copy commands (#2273,14037)
- document `VK_FORMAT_FEATURE_ACCELERATION_STRUCTURE_VERTEX_BUFFER_BIT_KHR`
- specify that acceleration structures are non-linear (#2289,14068)
- add several missing VUs for strides, vertexFormat, and indexType (#2315,14069)
- restore VUs for `VkAccelerationStructureBuildGeometryInfoKHR` (#2337,14098)
- ban multi-instance memory for host operations (#2324,14102)
- allow `dstAccelerationStructure` to be null for `vkGetAccelerationStructureBuildSizesKHR` (#2330,14111)
- more build VU cleanup (#2138,14130)
- specify host endianness for AS serialization (#2261,14136)
- add invertible transform matrix VU (#1710,14140)
- require `geometryCount` to be 1 for TLAS builds (#4145)
- improved validity conditions for build addresses (#4142)
- add single statement SPIR-V VUs, build limit VUs (#4158)
- document limits for vertex and aabb strides (#2390,14184)
- specify that `VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_KHR` applies to AS copies (#2382,14173)
- define sync for AS build inputs and indirect buffer (#2407,14208)

**VK_KHR_android_surface**

Name String

VK_KHR_android_surface

Extension Type

Instance extension

Registered Extension Number

9

Revision

6

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`
Contact

- Jesse Hall (critsec)

Other Extension Metadata

Last Modified Date

2016-01-14

IP Status

No known IP claims.

Contributors

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Ray Smith, ARM
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG

Description

The `VK_KHR_android_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to an `ANativeWindow`, Android's native surface type. The `ANativeWindow` represents the producer endpoint of any buffer queue, regardless of consumer endpoint. Common consumer endpoints for `ANativeWindows` are the system window compositor, video encoders, and application-specific compositors importing the images through a `SurfaceTexture`.

New Base Types

- ANativeWindow
New Commands

- `vkCreateAndroidSurfaceKHR`

New Structures

- `VkAndroidSurfaceCreateInfoKHR`

New Bitmasks

- `VkAndroidSurfaceCreateFlagsKHR`

New Enum Constants

- `VK_KHR_ANDROID_SURFACE_EXTENSION_NAME`
- `VK_KHR_ANDROID_SURFACE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR`

Issues

1) Does Android need a way to query for compatibility between a particular physical device (and queue family?) and a specific Android display?

**RESOLVED**: No. Currently on Android, any physical device is expected to be able to present to the system compositor, and all queue families must support the necessary image layout transitions and synchronization operations.

Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft.
- Revision 2, 2015-10-26 (Ian Elliott)
  - Renamed from VK_EXT_KHR_android_surface to VK_KHR_android_surface.
- Revision 3, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to surface creation function.
- Revision 4, 2015-11-10 (Jesse Hall)
  - Removed VK_ERROR_INVALID_ANDROID_WINDOW_KHR.
- Revision 5, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a pCreateInfo structure.
- Revision 6, 2016-01-14 (James Jones)
  - Moved VK_ERROR_NATIVE_WINDOW_IN_USE_KHR from the VK_KHR_android_surface to the VK_KHR_surface extension.
VK_KHR_buffer_device_address

Name String
VK_KHR_buffer_device_address

Extension Type
Device extension

Registered Extension Number
258

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Deprecation state
• Promoted to Vulkan 1.2

Contact
• Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

Other Extension Metadata

Last Modified Date
2019-06-24

IP Status
No known IP claims.

Interactions and External Dependencies
• Promoted to Vulkan 1.2 Core
• This extension requires SPV_KHR_physical_storage_buffer
• This extension provides API support for GL_EXT_buffer_reference and GL_EXT_buffer_reference2 and GL_EXT_buffer_reference_uvec2

Contributors
• Jeff Bolz, NVIDIA
• Neil Henning, AMD
• Tobias Hector, AMD
• Jason Ekstrand, Intel
• Baldur Karlsson, Valve
• Jan-Harald Fredriksen, Arm

Appendix E: Layers & Extensions (Informative) | 2521
Description

This extension allows the application to query a 64-bit buffer device address value for a buffer, which can be used to access the buffer memory via the `PhysicalStorageBuffer` storage class in the `GL_EXT_buffer_reference` GLSL extension and `SPV_KHR_physical_storage_buffer` SPIR-V extension.

Another way to describe this extension is that it adds “pointers to buffer memory in shaders”. By calling `vkGetBufferDeviceAddress` with a `VkBuffer`, it will return a `VkDeviceAddress` value which represents the address of the start of the buffer.

`vkGetBufferOpaqueCaptureAddress` and `vkGetDeviceMemoryOpaqueCaptureAddress` allow opaque addresses for buffers and memory objects to be queried for the current process. A trace capture and replay tool can then supply these addresses to be used at replay time to match the addresses used when the trace was captured. To enable tools to insert these queries, new memory allocation flags must be specified for memory objects that will be bound to buffers accessed via the `PhysicalStorageBuffer` storage class. Note that this mechanism is intended only to support capture/replay tools, and is not recommended for use in other applications.

There are various use cases this extension is designed for. It is required for ray tracing, useful for DX12 portability, and by allowing buffer addresses to be stored in memory it enables more complex data structures to be created.

This extension can also be used to hardcode a dedicated debug channel into all shaders by querying a pointer at startup and pushing that into shaders as a run-time constant (e.g. specialization constant) that avoids impacting other descriptor limits.

There are examples of usage in the `GL_EXT_buffer_reference` spec for how to use this in a high-level shading language such as GLSL. The `GL_EXT_buffer_reference2` and `GL_EXT_buffer_reference_uvec2` extensions were also added to help cover a few additional edge cases.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the `bufferDeviceAddress` capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- `vkGetBufferDeviceAddressKHR`
- `vkGetBufferOpaqueCaptureAddressKHR`
- `vkGetDeviceMemoryOpaqueCaptureAddressKHR`

New Structures

- `VkBufferDeviceAddressInfoKHR`
- `VkDeviceMemoryOpaqueCaptureAddressInfoKHR`
- Extending `VkBufferCreateInfo`:
- **VkBufferOpaqueCaptureAddressCreateInfoKHR**
  - Extending **VkMemoryAllocateInfo**:
    - **VkMemoryOpaqueCaptureAddressAllocateInfoKHR**
  - Extending **VkPhysicalDeviceFeatures2**, **VkDeviceCreateInfo**:
    - **VkPhysicalDeviceBufferDeviceAddressFeaturesKHR**

### New Enum Constants
- **VK_KHR_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME**
- **VK_KHR_BUFFER_DEVICE_ADDRESS_SPEC_VERSION**
- Extending **VkBufferCreateFlagBits**:
  - **VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR**
- Extending **VkBufferUsageFlagBits**:
  - **VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_KHR**
- Extending **VkMemoryAllocateFlagBits**:
  - **VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_BIT_KHR**
  - **VK_MEMORY_ALLOCATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_KHR**
- Extending **VkResult**:
  - **VK_ERROR_INVALID_OPAQUE_CAPTURE_ADDRESS_KHR**
- Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_BUFFER_DEVICE_ADDRESS_INFO_KHR**
  - **VK_STRUCTURE_TYPE_BUFFER_OPAQUE_CAPTURE_ADDRESS_CREATE_INFO_KHR**
  - **VK_STRUCTURE_TYPE_DEVICE_MEMORY_OPAQUE_CAPTURE_ADDRESS_INFO_KHR**
  - **VK_STRUCTURE_TYPE_MEMORY_OPAQUE_CAPTURE_ADDRESS_ALLOCATE_INFO_KHR**
  - **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_KHR**

### New SPIR-V Capabilities
- **PhysicalStorageBufferAddresses**

### Version History
- Revision 1, 2019-06-24 (Jan-Harald Fredriksen)
  - Internal revisions based on VK_EXT_buffer_device_address

**VK_KHR_copy_commands2**

**Name String**

*VK_KHR_copy_commands2*
Extension Type

Device extension

Registered Extension Number

338

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Jeff Leger @jackohound

Other Extension Metadata

Last Modified Date

2020-07-06

Interactions and External Dependencies

- None

Contributors

- Jeff Leger, Qualcomm
- Tobias Hector, AMD
- Jan-Harald Fredriksen, ARM
- Tom Olson, ARM

Description

This extension provides extensible versions of the Vulkan buffer and image copy commands. The new commands are functionally identical to the core commands, except that their copy parameters are specified using extensible structures that can be used to pass extension-specific information.

The following extensible copy commands are introduced with this extension: vkCmdCopyBuffer2KHR, vkCmdCopyImage2KHR, vkCmdCopyBufferToImage2KHR, vkCmdCopyImageToBuffer2KHR, vkCmdBlitImage2KHR, and vkCmdResolveImage2KHR. Each command contains an *Info2KHR structure parameter that includes sType/pNext members. Lower level structures that describe each region to be copied are also extended with sType/pNext members.

New Commands

- vkCmdBlitImage2KHR
- vkCmdCopyBuffer2KHR
- vkCmdCopyBufferToImage2KHR
• vkCmdCopyImage2KHR
• vkCmdCopyImageToBuffer2KHR
• vkCmdResolveImage2KHR

New Structures

• VkBlitImageInfo2KHR
• VkBufferCopy2KHR
• VkBufferImageCopy2KHR
• VkCopyBufferInfo2KHR
• VkCopyBufferToImageInfo2KHR
• VkCopyImageInfo2KHR
• VkCopyImageToBufferInfo2KHR
• VkImageBlit2KHR
• VkImageCopy2KHR
• VkImageResolve2KHR
• VkResolveImageInfo2KHR

New Enum Constants

• VK_KHR_COPY_COMMANDS_2_EXTENSION_NAME
• VK_KHR_COPY_COMMANDS_2_SPEC_VERSION

Extending VkStructureType:

◦ VK_STRUCTURE_TYPE_BLIT_IMAGE_INFO_2_KHR
◦ VK_STRUCTURE_TYPE_BUFFER_COPY_2_KHR
◦ VK_STRUCTURE_TYPE_BUFFER_IMAGE_COPY_2_KHR
◦ VK_STRUCTURE_TYPE_COPY_BUFFER_INFO_2_KHR
◦ VK_STRUCTURE_TYPE_COPY_BUFFER_TO_IMAGE_INFO_2_KHR
◦ VK_STRUCTURE_TYPE_COPY_IMAGE_INFO_2_KHR
◦ VK_STRUCTURE_TYPE_COPY_IMAGE_TO_BUFFER_INFO_2_KHR
◦ VK_STRUCTURE_TYPE_IMAGE_BLIT_2_KHR
◦ VK_STRUCTURE_TYPE_IMAGE_COPY_2_KHR
◦ VK_STRUCTURE_TYPE_IMAGE_RESOLVE_2_KHR
◦ VK_STRUCTURE_TYPE_RESOLVE_IMAGE_INFO_2_KHR

Version History

• Revision 1, 2020-07-06 (Jeff Leger)
VK_KHR_create_renderpass2

Name String
VK_KHR_create_renderpass2

Extension Type
Device extension

Registered Extension Number
110

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_multiview
• Requires VK_KHR_maintenance2

Deprecation state
• Promoted to Vulkan 1.2

Contact
• Tobias Hector Tobias

Other Extension Metadata

Last Modified Date
2018-02-07

Interactions and External Dependencies
• Promoted to Vulkan 1.2 Core

Contributors
• Tobias Hector
• Jeff Bolz

Description
This extension provides a new entry point to create render passes in a way that can be easily extended by other extensions through the substructures of render pass creation. The Vulkan 1.0 render pass creation sub-structures do not include sType/pNext members. Additionally, the renderpass begin/next/end commands have been augmented with new extensible structures for passing additional subpass information.
The `VkRenderPassMultiviewCreateInfo` and `VkInputAttachmentAspectReference` structures that extended the original ` VkRenderPassCreateInfo` are not accepted into the new creation functions, and instead their parameters are folded into this extension as follows:

- Elements of `VkRenderPassMultiviewCreateInfo::pViewMasks` are now specified in `VkSubpassDescription2KHR::viewMask`.
- Elements of `VkRenderPassMultiviewCreateInfo::pViewOffsets` are now specified in `VkSubpassDependency2KHR::viewOffset`.
- `VkRenderPassMultiviewCreateInfo::correlationMaskCount` and `VkRenderPassMultiviewCreateInfo::pCorrelationMasks` are directly specified in `VkRenderPassCreateInfo2KHR`.
- `VkInputAttachmentAspectReference::aspectMask` is now specified in the relevant input attachment description in `VkAttachmentDescription2KHR::aspectMask`.

The details of these mappings are explained fully in the new structures.

**Promotion to Vulkan 1.2**

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

**New Commands**

- `vkCmdBeginRenderPass2KHR`
- `vkCmdEndRenderPass2KHR`
- `vkCmdNextSubpass2KHR`
- `vkCreateRenderPass2KHR`

**New Structures**

- `VkAttachmentDescription2KHR`
- `VkAttachmentReference2KHR`
- `VkRenderPassCreateInfo2KHR`
- `VkSubpassBeginInfoKHR`
- `VkSubpassDependency2KHR`
- `VkSubpassDescription2KHR`
- `VkSubpassEndInfoKHR`

**New Enum Constants**

- `VK_KHR_CREATE_RENDERPASS_2_EXTENSION_NAME`
- `VK_KHR_CREATE_RENDERPASS_2_SPEC_VERSION`
- Extending `VkStructureType`:
Version History

- Revision 1, 2018-02-07 (Tobias Hector)
  - Internal revisions

VK_KHR_deferred_host_operations

Name String
  VK_KHR_deferred_host_operations

Extension Type
  Device extension

Registered Extension Number
  269

Revision
  4

Extension and Version Dependencies
  - Requires Vulkan 1.0

Contact
  - Josh Barczak (@jbarczak)

Other Extension Metadata

Last Modified Date
  2020-11-12

IP Status
  No known IP claims.

Contributors
  - Joshua Barczak, Intel
  - Jeff Bolz, NVIDIA
The \texttt{VK_KHR_deferred_host_operations} extension defines the infrastructure and usage patterns for deferrable commands, but does not specify any commands as deferrable. This is left to additional dependent extensions. Commands must not be deferred unless the deferral is specifically allowed by another extension which depends on \texttt{VK_KHR_deferred_host_operations}.

**New Object Types**

- \texttt{VkDeferredOperationKHR}

**New Commands**

- \texttt{vkCreateDeferredOperationKHR}
- \texttt{vkDeferredOperationJoinKHR}
- \texttt{vkDestroyDeferredOperationKHR}
- \texttt{vkGetDeferredOperationMaxConcurrencyKHR}
- \texttt{vkGetDeferredOperationResultKHR}

**New Enum Constants**

- \texttt{VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME}
- \texttt{VK_KHR_DEFERRED_HOST_OPERATIONS_SPEC_VERSION}

Extending \texttt{VkObjectType}:

- \texttt{VK_OBJECT_TYPE_DEFERRED_OPERATION_KHR}

Extending \texttt{VkResult}:

- \texttt{VK_OPERATION_DEFERRED_KHR}
- \texttt{VK_OPERATION_NOT_DEFERRED_KHR}
- \texttt{VK_THREAD_DONE_KHR}
- \texttt{VK_THREAD_IDLE_KHR}
- \texttt{VK_THREAD_IDLE_KHR}
Code Examples

The following examples will illustrate the concept of deferrable operations using a hypothetical example. The command `vkDoSomethingExpensiveEXT` denotes a deferrable command. The structure `VkExpensiveOperationArgsEXT` represents the arguments which it would normally accept.

The following example illustrates how a vulkan application might request deferral of an expensive operation:

```c
// create a deferred operation
VkDeferredOperationKHR hOp;
VkResult result = vkCreateDeferredOperationKHR(device, pCallbacks, &hOp);
assert(result == VK_SUCCESS);

result = vkDoSomethingExpensive(device, hOp, ...);
assert(result == VK_OPERATION_DEFERRED_KHR);

// operation was deferred. Execute it asynchronously
std::async::launch(
    [ hOp ] ( )
    {
        vkDeferredOperationJoinKHR(device, hOp);
        result = vkGetDeferredOperationResultKHR(device, hOp);
        // deferred operation is now complete. 'result' indicates success or failure
        vkDestroyDeferredOperationKHR(device, hOp, pCallbacks);
    }
);
```

The following example shows a subroutine which guarantees completion of a deferred operation, in the presence of multiple worker threads, and returns the result of the operation.
VkResult FinishDeferredOperation(VkDeferredOperationKHR hOp)
{
    // Attempt to join the operation until the implementation indicates that we should stop

    VkResult result = vkDeferredOperationJoinKHR(device, hOp);
    while( result == VK_THREAD_IDLE_KHR )
    {
        std::this_thread::yield();
        result = vkDeferredOperationJoinKHR(device, hOp);
    }

    switch( result )
    {
    case VK_SUCCESS:
        { // deferred operation has finished. Query its result
            result = vkGetDeferredOperationResultKHR(device, hOp);
        }
        break;

    case VK_THREAD_DONE_KHR:
        { // deferred operation is being wrapped up by another thread
            // wait for that thread to finish
            do
            {
                std::this_thread::yield();
                result = vkGetDeferredOperationResultKHR(device, hOp);
            } while( result == VK_NOT_READY );
        }
        break;

    default:
        assert(false); // other conditions are illegal.
        break;
    }

    return result;
}

Issues

1. Should this extension have a VkPhysicalDevice*FeaturesKHR structure?

RESOLVED: No. This extension does not add any functionality on its own and requires a dependent extension to actually enable functionality and thus there is no value in adding a feature structure. If necessary, any dependent extension could add a feature boolean if it wanted to indicate that it is adding optional deferral support.
Version History

- Revision 1, 2019-12-05 (Josh Barczak, Daniel Koch)
  - Initial draft.
- Revision 2, 2020-03-06 (Daniel Koch, Tobias Hector)
  - Add missing VK_OBJECT_TYPE_DEFERRED_OPERATION_KHR enum
  - Fix sample code
  - Clarified deferred operation parameter lifetimes (#2018,!3647)
- Revision 3, 2020-05-15 (Josh Barczak)
  - Clarify behavior of vkGetDeferredOperationMaxConcurrencyKHR, allowing it to return 0 if the operation is complete (#2036,!3850)
- Revision 4, 2020-11-12 (Tobias Hector, Daniel Koch)
  - Remove VkDeferredOperationInfoKHR and change return value semantics when deferred host operations are in use (#2067,3813)
  - Clarify return value of vkGetDeferredOperationResultKHR (#2339,!4110)

VK_KHR_depth_stencil_resolve

Name String

   VK_KHR_depth_stencil_resolve

Extension Type

   Device extension

Registered Extension Number

   200

Revision

   1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_create_renderpass2

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Jan-Harald Fredriksen janharald

Other Extension Metadata

Last Modified Date

   2018-04-09
Interactions and External Dependencies

- Promoted to Vulkan 1.2 Core

Contributors

- Jan-Harald Fredriksen, Arm
- Andrew Garrard, Samsung Electronics
- Soowan Park, Samsung Electronics
- Jeff Bolz, NVIDIA
- Daniel Rakos, AMD

Description

This extension adds support for automatically resolving multisampled depth/stencil attachments in a subpass in a similar manner as for color attachments.

Multisampled color attachments can be resolved at the end of a subpass by specifying $p\text{ResolveAttachments}$ entries corresponding to the $p\text{ColorAttachments}$ array entries. This does not allow for a way to map the resolve attachments to the depth/stencil attachment. The $\text{vkCmdResolveImage}$ command does not allow for depth/stencil images. While there are other ways to resolve the depth/stencil attachment, they can give sub-optimal performance. Extending the $\text{VkSubpassDescription2}$ in this extension allows an application to add a $p\text{DepthStencilResolveAttachment}$, that is similar to the color $p\text{ResolveAttachments}$, that the $p\text{DepthStencilAttachment}$ can be resolved into.

Depth and stencil samples are resolved to a single value based on the resolve mode. The set of possible resolve modes is defined in the $\text{VkResolveModeFlagBits}$ enum. The $\text{VK\_RESOLVE\_MODE\_SAMPLE\_ZERO\_BIT}$ mode is the only mode that is required of all implementations (that support the extension or support Vulkan 1.2 or higher). Some implementations may also support averaging (the same as color sample resolve) or taking the minimum or maximum sample, which may be more suitable for depth/stencil resolve.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending $\text{VkPhysicalDeviceProperties2}$:
  - $\text{VkPhysicalDeviceDepthStencilResolvePropertiesKHR}$
- Extending $\text{VkSubpassDescription2}$:
  - $\text{VkSubpassDescriptionDepthStencilResolveKHR}$

New Enums

- $\text{VkResolveModeFlagBitsKHR}$
New Bitmasks

- VkResolveModeFlagsKHR

New Enum Constants

- VK_KHR_DEPTH_STENCIL_RESOLVE_EXTENSION_NAME
- VK_KHR_DEPTH_STENCIL_RESOLVE_SPEC_VERSION

Extending VkResolveModeFlagBits:

- VK_RESOLVE_MODE_AVERAGE_BIT_KHR
- VK_RESOLVE_MODE_MAX_BIT_KHR
- VK_RESOLVE_MODE_MIN_BIT_KHR
- VK_RESOLVE_MODE_NONE_KHR
- VK_RESOLVE_MODE_SAMPLE_ZERO_BIT_KHR

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_STENCIL_RESOLVE_PROPERTIES_KHR
- VK_STRUCTURE_TYPE_SUBPASS_DESCRIPTION_DEPTH_STENCIL_RESOLVE_KHR

Version History

- Revision 1, 2018-04-09 (Jan-Harald Fredriksen)
  - Initial revision

VK_KHR_display

Name String

VK_KHR_display

Extension Type

Instance extension

Registered Extension Number

3

Revision

23

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_surface

Contact

- James Jones @cubanismo
Other Extension Metadata

Last Modified Date
2017-03-13

IP Status
No known IP claims.

Contributors
• James Jones, NVIDIA
• Norbert Nopper, Freescale
• Jeff Vigil, Qualcomm
• Daniel Rakos, AMD

Description
This extension provides the API to enumerate displays and available modes on a given device.

New Object Types
• VkDisplayKHR
• VkDisplayModeKHR

New Commands
• vkCreateDisplayModeKHR
• vkCreateDisplayPlaneSurfaceKHR
• vkGetDisplayModePropertiesKHR
• vkGetDisplayPlaneCapabilitiesKHR
• vkGetDisplayPlaneSupportedDisplaysKHR
• vkGetPhysicalDeviceDisplayPlanePropertiesKHR
• vkGetPhysicalDeviceDisplayPropertiesKHR

New Structures
• VkDisplayModeCreateInfoKHR
• VkDisplayModeParametersKHR
• VkDisplayModePropertiesKHR
• VkDisplayPlaneCapabilitiesKHR
• VkDisplayPlanePropertiesKHR
New Enums

- VkDisplayPlaneAlphaFlagBitsKHR

New Bitmasks

- VkDisplayModeCreateFlagsKHR
- VkDisplayPlaneAlphaFlagsKHR
- VkDisplaySurfaceCreateFlagsKHR
- VkSurfaceTransformFlagsKHR

New Enum Constants

- VK_KHR_DISPLAY_EXTENSION_NAME
- VK_KHR_DISPLAY_SPEC_VERSION

Extending VkObjectType:
- VK_OBJECT_TYPE_DISPLAY_KHR
- VK_OBJECT_TYPE_DISPLAY_MODE_KHR

Extending VkStructureType:
- VK_STRUCTURE_TYPE_DISPLAY_MODE_CREATE_INFO_KHR
- VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR

Issues

1) Which properties of a mode should be fixed in the mode info vs. settable in some other function when setting the mode? E.g., do we need to double the size of the mode pool to include both stereo and non-stereo modes? YUV and RGB scanout even if they both take RGB input images? BGR vs. RGB input? etc.

**PROPOSED RESOLUTION:** Many modern displays support at most a handful of resolutions and timings natively. Other “modes” are expected to be supported using scaling hardware on the display engine or GPU. Other properties, such as rotation and mirroring should not require duplicating hardware modes just to express all combinations. Further, these properties may be implemented on a per-display or per-overlay granularity.

To avoid the exponential growth of modes as mutable properties are added, as was the case with EGLConfig/WGL pixel formats/GLXFBConfig, this specification should separate out hardware properties and configurable state into separate objects. Modes and overlay planes will express capabilities of the hardware, while a separate structure will allow applications to configure scaling, rotation, mirroring, color keys, LUT values, alpha masks, etc. for a given swapchain independent of the mode in use. Constraints on these settings will be established by properties of the immutable...
objects.

Note the resolution of this issue may affect issue 5 as well.

2) What properties of a display itself are useful?

**PROPOSED RESOLUTION**: This issue is too broad. It was meant to prompt general discussion, but resolving this issue amounts to completing this specification. All interesting properties should be included. The issue will remain as a placeholder since removing it would make it hard to parse existing discussion notes that refer to issues by number.

3) How are multiple overlay planes within a display or mode enumerated?

**PROPOSED RESOLUTION**: They are referred to by an index. Each display will report the number of overlay planes it contains.

4) Should swapchains be created relative to a mode or a display?

**PROPOSED RESOLUTION**: When using this extension, swapchains are created relative to a mode and a plane. The mode implies the display object the swapchain will present to. If the specified mode is not the display's current mode, the new mode will be applied when the first image is presented to the swapchain, and the default operating system mode, if any, will be restored when the swapchain is destroyed.

5) Should users query generic ranges from displays and construct their own modes explicitly using those constraints rather than querying a fixed set of modes (Most monitors only have one real “mode” these days, even though many support relatively arbitrary scaling, either on the monitor side or in the GPU display engine, making “modes” something of a relic/compatibility construct).

**PROPOSED RESOLUTION**: Expose both. Display info structures will expose a set of predefined modes, as well as any attributes necessary to construct a customized mode.

6) Is it fine if we return the display and display mode handles in the structure used to query their properties?

**PROPOSED RESOLUTION**: Yes.

7) Is there a possibility that not all displays of a device work with all of the present queues of a device? If yes, how do we determine which displays work with which present queues?

**PROPOSED RESOLUTION**: No known hardware has such limitations, but determining such limitations is supported automatically using the existing `VK_KHR_surface` and `VK_KHR_swapchain` query mechanisms.

8) Should all presentation need to be done relative to an overlay plane, or can a display mode + display be used alone to target an output?

**PROPOSED RESOLUTION**: Require specifying a plane explicitly.

9) Should displays have an associated window system display, such as an `HDC` or `Display`?

**PROPOSED RESOLUTION**: No. Displays are independent of any windowing system in use on the
system. Further, neither HDC nor Display* refer to a physical display object.

10) Are displays queried from a physical GPU or from a device instance?

**PROPOSED RESOLUTION:** Developers prefer to query modes directly from the physical GPU so they can use display information as an input to their device selection algorithms prior to device creation. This avoids the need to create placeholder device instances to enumerate displays.

This preference must be weighed against the extra initialization that must be done by driver vendors prior to device instance creation to support this usage.

11) Should displays and/or modes be dispatchable objects? If functions are to take displays, overlays, or modes as their first parameter, they must be dispatchable objects as defined in Khronos bug 13529. If they are not added to the list of dispatchable objects, functions operating on them must take some higher-level object as their first parameter. There is no performance case against making them dispatchable objects, but they would be the first extension objects to be dispatchable.

**PROPOSED RESOLUTION:** Do not make displays or modes dispatchable. They will dispatch based on their associated physical device.

12) Should hardware cursor capabilities be exposed?

**PROPOSED RESOLUTION:** Defer. This could be a separate extension on top of the base WSI specs.

if they are one physical display device to an end user, but may internally be implemented as two side-by-side displays using the same display engine (and sometimes cabling) resources as two physically separate display devices.

**RESOLVED:** Tiled displays will appear as a single display object in this API.

14) Should the raw EDID data be included in the display information?

**RESOLVED:** No. A future extension could be added which reports the EDID if necessary. This may be complicated by the outcome of issue 13.

15) Should min and max scaling factor capabilities of overlays be exposed?

**RESOLVED:** Yes. This is exposed indirectly by allowing applications to query the min/max position and extent of the source and destination regions from which image contents are fetched by the display engine when using a particular mode and overlay pair.

16) Should devices be able to expose planes that can be moved between displays? If so, how?

**RESOLVED:** Yes. Applications can determine which displays a given plane supports using vkGetDisplayPlaneSupportedDisplaysKHR.

17) Should there be a way to destroy display modes? If so, does it support destroying “built in” modes?

**RESOLVED:** Not in this extension. A future extension could add this functionality.
18) What should the lifetime of display and built-in display mode objects be?

**RESOLVED:** The lifetime of the instance. These objects cannot be destroyed. A future extension may be added to expose a way to destroy these objects and/or support display hotplug.

19) Should persistent mode for smart panels be enabled/disabled at swapchain creation time, or on a per-present basis.

**RESOLVED:** On a per-present basis.

**Examples**

Note

The example code for the `VK_KHR_display` and `VK_KHR_display_swapchain` extensions was removed from the appendix after revision 1.0.43. The display enumeration example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: [https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c](https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c)).

**Version History**

- Revision 1, 2015-02-24 (James Jones)
  - Initial draft
- Revision 2, 2015-03-12 (Norbert Nopper)
  - Added overlay enumeration for a display.
- Revision 3, 2015-03-17 (Norbert Nopper)
  - Fixed typos and namings as discussed in Bugzilla.
  - Reordered and grouped functions.
  - Added functions to query count of display, mode and overlay.
  - Added native display handle, which may be needed on some platforms to create a native Window.
- Revision 4, 2015-03-18 (Norbert Nopper)
  - Removed primary and virtualPosition members (see comment of James Jones in Bugzilla).
  - Added native overlay handle to info structure.
  - Replaced , with ; in struct.
- Revision 6, 2015-03-18 (Daniel Rakos)
  - Added WSI extension suffix to all items.
  - Made the whole API more “Vulkanish”.
  - Replaced all functions with a single `vkGetDisplayInfoKHR` function to better match the rest of the API.
  - Made the display, display mode, and overlay objects be first class objects, not subclasses of
VkBaseObject as they do not support the common functions anyways.

- Renamed *Info structures to *Properties.
- Removed overlayIndex field from VkOverlayProperties as there is an implicit index already as a result of moving to a “Vulkanish” API.
- Displays are not get through device, but through physical GPU to match the rest of the Vulkan API. Also this is something ISVs explicitly requested.
- Added issue (6) and (7).

**Revision 7, 2015-03-25 (James Jones)**

- Added an issues section
- Added rotation and mirroring flags

**Revision 8, 2015-03-25 (James Jones)**

- Combined the duplicate issues sections introduced in last change.
- Added proposed resolutions to several issues.

**Revision 9, 2015-04-01 (Daniel Rakos)**

- Rebased extension against Vulkan 0.82.0

**Revision 10, 2015-04-01 (James Jones)**

- Added issues (10) and (11).
- Added more straw-man issue resolutions, and cleaned up the proposed resolution for issue (4).
- Updated the rotation and mirroring enums to have proper bitmask semantics.

**Revision 11, 2015-04-15 (James Jones)**

- Added proposed resolution for issues (1) and (2).
- Added issues (12), (13), (14), and (15)
- Removed pNativeHandle field from overlay structure.
- Fixed small compilation errors in example code.

**Revision 12, 2015-07-29 (James Jones)**

- Rewrote the guts of the extension against the latest WSI swapchain specifications and the latest Vulkan API.
- Address overlay planes by their index rather than an object handle and refer to them as “planes” rather than “overlays” to make it slightly clearer that even a display with no “overlays” still has at least one base “plane” that images can be displayed on.
- Updated most of the issues.
- Added an “extension type” section to the specification header.
- Re-used the VK_EXT_KHR_surface surface transform enumerations rather than redefining them here.
- Updated the example code to use the new semantics.

**Revision 13, 2015-08-21 (Ian Elliott)**
• Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
• Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.

• Revision 14, 2015-09-01 (James Jones)
  ◦ Restore single-field revision number.

• Revision 15, 2015-09-08 (James Jones)
  ◦ Added alpha flags enum.
  ◦ Added premultiplied alpha support.

• Revision 16, 2015-09-08 (James Jones)
  ◦ Added description section to the spec.
  ◦ Added issues 16 - 18.

• Revision 17, 2015-10-02 (James Jones)
  ◦ Planes are now a property of the entire device rather than individual displays. This allows planes to be moved between multiple displays on devices that support it.
  ◦ Added a function to create a VkSurfaceKHR object describing a display plane and mode to align with the new per-platform surface creation conventions.
  ◦ Removed detailed mode timing data. It was agreed that the mode extents and refresh rate are sufficient for current use cases. Other information could be added back in as an extension if it is needed in the future.
  ◦ Added support for smart/persistent/buffered display devices.

• Revision 18, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_display to VK_KHR_display.

• Revision 19, 2015-11-02 (James Jones)
  ◦ Updated example code to match revision 17 changes.

• Revision 20, 2015-11-03 (Daniel Rakos)
  ◦ Added allocation callbacks to creation functions.

• Revision 21, 2015-11-10 (Jesse Hall)
  ◦ Added VK_DISPLAY_PLANE_ALPHA_OPAQUE_BIT_KHR, and use VkDisplayPlaneAlphaFlagBitsKHR for VkDisplayPlanePropertiesKHR::alphaMode instead of VkDisplayPlaneAlphaFlagsKHR, since it only represents one mode.
  ◦ Added reserved flags bitmask to VkDisplayPlanePropertiesKHR.
  ◦ Use VkSurfaceTransformFlagBitsKHR instead of obsolete VkSurfaceTransformKHR.
  ◦ Renamed vkGetDisplayPlaneSupportedDisplaysKHR parameters for clarity.

• Revision 22, 2015-12-18 (James Jones)
  ◦ Added missing “planeIndex” parameter to vkGetDisplayPlaneSupportedDisplaysKHR()

• Revision 23, 2017-03-13 (James Jones)
Closed all remaining issues. The specification and implementations have been shipping with the proposed resolutions for some time now.

Removed the sample code and noted it has been integrated into the official Vulkan SDK cube demo.

**VK_KHR_display_swapchain**

**Name String**

VK_KHR_display_swapchain

**Extension Type**

Device extension

**Registered Extension Number**

4

**Revision**

10

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_swapchain
- Requires VK_KHR_display

**Contact**

- James Jones cubanismo

**Other Extension Metadata**

**Last Modified Date**

2017-03-13

**IP Status**

No known IP claims.

**Contributors**

- James Jones, NVIDIA
- Jeff Vigil, Qualcomm
- Jesse Hall, Google

**Description**

This extension provides an API to create a swapchain directly on a device's display without any underlying window system.
New Commands

- vkCreateSharedSwapchainsKHR

New Structures

- Extending VkPresentInfoKHR:
  - VkDisplayPresentInfoKHR

New Enum Constants

- VK_KHR_DISPLAY_SWAPCHAIN_EXTENSION_NAME
- VK_KHR_DISPLAY_SWAPCHAIN_SPEC_VERSION

- Extending VkResult:
  - VK_ERROR_INCOMPATIBLE_DISPLAY_KHR

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DISPLAY_PRESENT_INFO_KHR

Issues

1) Should swapchains sharing images each hold a reference to the images, or should it be up to the application to destroy the swapchains and images in an order that avoids the need for reference counting?

**RESOLVED:** Take a reference. The lifetime of presentable images is already complex enough.

2) Should the srcRect and dstRect parameters be specified as part of the presentation command, or at swapchain creation time?

**RESOLVED:** As part of the presentation command. This allows moving and scaling the image on the screen without the need to respecify the mode or create a new swapchain and presentable images.

3) Should srcRect and dstRect be specified as rects, or separate offset/extent values?

**RESOLVED:** As rects. Specifying them separately might make it easier for hardware to expose support for one but not the other, but in such cases applications must just take care to obey the reported capabilities and not use non-zero offsets or extents that require scaling, as appropriate.

4) How can applications create multiple swapchains that use the same images?

**RESOLVED:** By calling vkCreateSharedSwapchainsKHR.

An earlier resolution used vkCreateSwapchainKHR, chaining multiple VkSwapchainCreateInfoKHR structures through pNext. In order to allow each swapchain to also allow other extension structs, a level of indirection was used: VkSwapchainCreateInfoKHR::pNext pointed to a different structure, which had both sType and pNext members for additional extensions, and also had a pointer to the next VkSwapchainCreateInfoKHR structure. The number of swapchains to be created could only be found by walking this linked list of alternating structures, and the pSwapchains out parameter was
reinterpreted to be an array of VkSwapchainKHR handles.

Another option considered was a method to specify a “shared” swapchain when creating a new swapchain, such that groups of swapchains using the same images could be built up one at a time. This was deemed unusable because drivers need to know all of the displays an image will be used on when determining which internal formats and layouts to use for that image.

Examples

Note

The example code for the VK_KHR_display and VK_KHR_display_swapchain extensions was removed from the appendix after revision 1.0.43. The display swapchain creation example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c).

Version History

- Revision 1, 2015-07-29 (James Jones)
  ◦ Initial draft
- Revision 2, 2015-08-21 (Ian Elliott)
  ◦ Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  ◦ Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.
- Revision 3, 2015-09-01 (James Jones)
  ◦ Restore single-field revision number.
- Revision 4, 2015-09-08 (James Jones)
  ◦ Allow creating multiple swap chains that share the same images using a single call to vkCreateSwapChainKHR().
- Revision 5, 2015-09-10 (Alon Or-bach)
  ◦ Removed underscores from SWAP_CHAIN in two enums.
- Revision 6, 2015-10-02 (James Jones)
  ◦ Added support for smart panels/buffered displays.
- Revision 7, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_display_swapchain to VK_KHR_display_swapchain.
- Revision 8, 2015-11-03 (Daniel Rakos)
  ◦ Updated sample code based on the changes to VK_KHR_swapchain.
- Revision 9, 2015-11-10 (Jesse Hall)
  ◦ Replaced VkDisplaySwapchainCreateInfoKHR with vkCreateSharedSwapchainsKHR, changing resolution of issue #4.
• Revision 10, 2017-03-13 (James Jones)
  ◦ Closed all remaining issues. The specification and implementations have been shipping with
    the proposed resolutions for some time now.
  ◦ Removed the sample code and noted it has been integrated into the official Vulkan SDK cube
demo.

**VK_KHR_draw_indirect_count**

**Name String**

VK_KHR_draw_indirect_count

**Extension Type**

Device extension

**Registered Extension Number**

170

**Revision**

1

**Extension and Version Dependencies**

• Requires Vulkan 1.0

**Deprecation state**

• Promoted to Vulkan 1.2

**Contact**

• Piers Daniell [pdaniell-nv](mailto:pdaniell-nv)

**Other Extension Metadata**

**Last Modified Date**

2017-08-25

**Interactions and External Dependencies**

• Promoted to Vulkan 1.2 Core

**IP Status**

No known IP claims.

**Contributors**

• Matthaeus G. Chajdas, AMD
• Derrick Owens, AMD
• Graham Sellers, AMD
• Daniel Rakos, AMD
• Dominik Witczak, AMD
Description

This extension is based off the VK_AMD_draw_indirect_count extension. This extension allows an application to source the number of draw calls for indirect draw calls from a buffer.

Applications might want to do culling on the GPU via a compute shader prior to the draw. This enables the application to generate arbitrary amounts of draw commands and execute them without host intervention.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the entry points vkCmdDrawIndirectCount and vkCmdDrawIndexedIndirectCount are optional. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- vkCmdDrawIndexedIndirectCountKHR
- vkCmdDrawIndirectCountKHR

New Enum Constants

- VK_KHR_DRAW_INDIRECT_COUNT_EXTENSION_NAME
- VK_KHR_DRAW_INDIRECT_COUNT_SPEC_VERSION

Version History

- Revision 1, 2017-08-25 (Piers Daniell)
  - Initial draft based off VK_AMD_draw_indirect_count

VK_KHR_driver_properties

Name String

VK_KHR_driver_properties

Extension Type

Device extension

Registered Extension Number

197

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Daniel Rakos @drakos-amd

Other Extension Metadata

Last Modified Date

2018-04-11

Interactions and External Dependencies

- Promoted to Vulkan 1.2 Core

IP Status

No known IP claims.

Contributors

- Baldur Karlsson
- Matthaeus G. Chajdas, AMD
- Piers Daniell, NVIDIA
- Alexander Galazin, Arm
- Jesse Hall, Google
- Daniel Rakos, AMD

Description

This extension provides a new physical device query which allows retrieving information about the driver implementation, allowing applications to determine which physical device corresponds to which particular vendor's driver, and which conformance test suite version the driver implementation is compliant with.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- `VkConformanceVersionKHR`
- Extending `VkPhysicalDeviceProperties2`
New Enums

- VkDriverIdKHR

New Enum Constants

- VK_KHR_DRIVER_PROPERTIES_EXTENSION_NAME
- VK_KHR_DRIVER_PROPERTIES_SPEC_VERSION
- VK_MAX_DRIVER_INFO_SIZE_KHR
- VK_MAX_DRIVER_NAME_SIZE_KHR

Extending VkDriverId:

- VK_DRIVER_ID_AMD_OPEN_SOURCE_KHR
- VK_DRIVER_ID_AMD_PROPRIETARY_KHR
- VK_DRIVER_ID_ARM_PROPRIETARY_KHR
- VK_DRIVER_ID_BROADCOM_PROPRIETARY_KHR
- VK_DRIVER_ID_GGP_PROPRIETARY_KHR
- VK_DRIVER_ID_GOOGLE_SWIFTSHADER_KHR
- VK_DRIVER_ID_IMAGINATION_PROPRIETARY_KHR
- VK_DRIVER_ID_INTEL_OPEN_SOURCE_MESA_KHR
- VK_DRIVER_ID_INTEL_PROPRIETARY_WINDOWS_KHR
- VK_DRIVER_ID_MESA_RADV_KHR
- VK_DRIVER_ID_NVIDIA_PROPRIETARY_KHR
- VK_DRIVER_ID_QUALCOMM_PROPRIETARY_KHR

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES_KHR

Version History

- Revision 1, 2018-04-11 (Daniel Rakos)
  - Internal revisions

VK_KHR_external_fence_fd

Name String

VK_KHR_external_fence_fd

Extension Type

Device extension
An application using external memory may wish to synchronize access to that memory using fences. This extension enables an application to export fence payload to and import fence payload from POSIX file descriptors.

**New Commands**

- `vkGetFenceFdKHR`
- `vkImportFenceFdKHR`

**New Structures**

- `VkFenceGetFdInfoKHR`
- `VkImportFenceFdInfoKHR`
New Enum Constants

- `VK_KHR_EXTERNAL_FENCE_FD_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_FENCE_FD_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_FENCE_GET_FD_INFO_KHR`
  - `VK_STRUCTURE_TYPE_IMPORT_FENCE_FD_INFO_KHR`

Issues

This extension borrows concepts, semantics, and language from `VK_KHR_external_semaphore_fd`. That extension's issues apply equally to this extension.

Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

`VK_KHR_external_fence_win32`

Name String

`VK_KHR_external_fence_win32`

Extension Type

Device extension

Registered Extension Number

115

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_external_fence`

Contact

- Jesse Hall @critsec

Other Extension Metadata

Last Modified Date

2017-05-08

IP Status

No known IP claims.
Description
An application using external memory may wish to synchronize access to that memory using
fences. This extension enables an application to export fence payload to and import fence payload
from Windows handles.

New Commands
- `vkGetFenceWin32HandleKHR`
- `vkImportFenceWin32HandleKHR`

New Structures
- `VkFenceGetWin32HandleInfoKHR`
- `VkImportFenceWin32HandleInfoKHR`
- Extending `VkFenceCreateInfo`:
  - `VkExportFenceWin32HandleInfoKHR`

New Enum Constants
- `VK_KHR_EXTERNAL_FENCE_WIN32_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_FENCE_WIN32_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_EXPORT_FENCE_WIN32_HANDLE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_FENCE_GET_WIN32_HANDLE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_IMPORT_FENCE_WIN32_HANDLE_INFO_KHR`

Issues
This extension borrows concepts, semantics, and language from `VK_KHR_external_semaphore_win32`. That extension’s issues apply equally to this extension.

1) Should D3D12 fence handle types be supported, like they are for semaphores?

**RESOLVED**: No. Doing so would require extending the fence signal and wait operations to provide values to signal / wait for, like `VkD3D12FenceSubmitInfoKHR` does. A D3D12 fence can be signaled by importing it into a `VkSemaphore` instead of a `VkFence`, and applications can check status or wait on
the D3D12 fence using non-Vulkan APIs. The convenience of being able to do these operations on
\texttt{VkFence} objects doesn't justify the extra API complexity.

\textbf{Version History}

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

\textbf{VK\_KHR\_external\_memory\_fd}

\textbf{Name String}

\texttt{VK\__KHR\_external\_memory\_fd}

\textbf{Extension Type}

Device extension

\textbf{Registered Extension Number}

75

\textbf{Revision}

1

\textbf{Extension and Version Dependencies}

- Requires Vulkan 1.0
- Requires \texttt{VK\_KHR\_external\_memory}

\textbf{Contact}

- James Jones \texttt{cubanismo}

\textbf{Other Extension Metadata}

\textbf{Last Modified Date}

2016-10-21

\textbf{IP Status}

No known IP claims.

\textbf{Contributors}

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA

\textbf{Description}

An application may wish to reference device memory in multiple Vulkan logical devices or
instances, in multiple processes, and/or in multiple APIs. This extension enables an application to
export POSIX file descriptor handles from Vulkan memory objects and to import Vulkan memory
objects from POSIX file descriptor handles exported from other Vulkan memory objects or from
similar resources in other APIs.

New Commands

- vkGetMemoryFdKHR
- vkGetMemoryFdPropertiesKHR

New Structures

- VkMemoryFdPropertiesKHR
- VkMemoryGetFdInfoKHR
- Extending VkMemoryAllocateInfo:
  - VkImportMemoryFdInfoKHR

New Enum Constants

- VK_KHR_EXTERNAL_MEMORY_FD_EXTENSION_NAME
- VK_KHR_EXTERNAL_MEMORY_FD_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMPORT_MEMORY_FD_INFO_KHR
  - VK_STRUCTURE_TYPE_MEMORY_FD_PROPERTIES_KHR
  - VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR

Issues

1) Does the application need to close the file descriptor returned by vkGetMemoryFdKHR?

**RESOLVED:** Yes, unless it is passed back in to a driver instance to import the memory. A successful get call transfers ownership of the file descriptor to the application, and a successful import transfers it back to the driver. Destroying the original memory object will not close the file descriptor or remove its reference to the underlying memory resource associated with it.

2) Do drivers ever need to expose multiple file descriptors per memory object?

**RESOLVED:** No. This would indicate there are actually multiple memory objects, rather than a single memory object.

3) How should the valid size and memory type for POSIX file descriptor memory handles created outside of Vulkan be specified?

**RESOLVED:** The valid memory types are queried directly from the external handle. The size will be specified by future extensions that introduce such external memory handle types.

Version History

- Revision 1, 2016-10-21 (James Jones)
VK_KHR_external_memory_win32

Name String
VK_KHR_external_memory_win32

Extension Type
Device extension

Registered Extension Number
74

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_external_memory

Contact
• James Jones (@cubanismo)

Other Extension Metadata

Last Modified Date
2016-10-21

IP Status
No known IP claims.

Contributors
• James Jones, NVIDIA
• Jeff Juliano, NVIDIA
• Carsten Rohde, NVIDIA

Description
An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension enables an application to export Windows handles from Vulkan memory objects and to import Vulkan memory objects from Windows handles exported from other Vulkan memory objects or from similar resources in other APIs.

New Commands
• vkGetMemoryWin32HandleKHR
• vkGetMemoryWin32HandlePropertiesKHR

New Structures

• VkMemoryGetWin32HandleInfoKHR
• VkMemoryWin32HandlePropertiesKHR
• Extending VkMemoryAllocateInfo:
  ◦ VkExportMemoryWin32HandleInfoKHR
  ◦ VkImportMemoryWin32HandleInfoKHR

New Enum Constants

• VK_KHR_EXTERNAL_MEMORY_WIN32_EXTENSION_NAME
• VK_KHR_EXTERNAL_MEMORY_WIN32_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_MEMORY_WIN32_HANDLE_PROPERTIES_KHR

Issues

1) Do applications need to call CloseHandle() on the values returned from vkGetMemoryWin32HandleKHR when handleType is VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR?

**RESOLVED:** Yes, unless it is passed back in to another driver instance to import the object. A successful get call transfers ownership of the handle to the application. Destroying the memory object will not destroy the handle or the handle’s reference to the underlying memory resource.

2) Should the language regarding KMT/Windows 7 handles be moved to a separate extension so that it can be deprecated over time?

**RESOLVED:** No. Support for them can be deprecated by drivers if they choose, by no longer returning them in the supported handle types of the instance level queries.

3) How should the valid size and memory type for windows memory handles created outside of Vulkan be specified?

**RESOLVED:** The valid memory types are queried directly from the external handle. The size is determined by the associated image or buffer memory requirements for external handle types that require dedicated allocations, and by the size specified when creating the object from which the handle was exported for other external handle types.
**Version History**

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

**VK_KHR_external_semaphore_fd**

**Name String**

VK_KHR_external_semaphore_fd

**Extension Type**

Device extension

**Registered Extension Number**

80

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_external_semaphore

**Contact**

- James Jones @cubanismo

**Other Extension Metadata**

**Last Modified Date**

2016-10-21

**IP Status**

No known IP claims.

**Contributors**

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

**Description**

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore payload to and import semaphore payload from POSIX file descriptors.
New Commands

- vkGetSemaphoreFdKHR
- vkImportSemaphoreFdKHR

New Structures

- VkImportSemaphoreFdInfoKHR
- VkSemaphoreGetFdInfoKHR

New Enum Constants

- VK_KHR_EXTERNAL_SEMAPHORE_FD_EXTENSION_NAME
- VK_KHR_EXTERNAL_SEMAPHORE_FD_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_FD_INFO_KHR
  - VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR

Issues

1) Does the application need to close the file descriptor returned by vkGetSemaphoreFdKHR?

**RESOLVED**: Yes, unless it is passed back in to a driver instance to import the semaphore. A successful get call transfers ownership of the file descriptor to the application, and a successful import transfers it back to the driver. Destroying the original semaphore object will not close the file descriptor or remove its reference to the underlying semaphore resource associated with it.

Version History

- Revision 1, 2016-10-21 (Jesse Hall)
  - Initial revision

**VK_KHR_external_semaphore_win32**

Name String

- VK_KHR_external_semaphore_win32

Extension Type

- Device extension

Registered Extension Number

- 79

Revision

- 1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_external_semaphore

Contact

- James Jones cubanismo

Other Extension Metadata

Last Modified Date
2016-10-21

IP Status
No known IP claims.

Contributors

- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Carsten Rohde, NVIDIA

Description

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore payload to and import semaphore payload from Windows handles.

New Commands

- vkGetSemaphoreWin32HandleKHR
- vkImportSemaphoreWin32HandleKHR

New Structures

- VkImportSemaphoreWin32HandleInfoKHR
- VkSemaphoreGetWin32HandleInfoKHR
- Extending VkSemaphoreCreateInfo:
  - VkExportSemaphoreWin32HandleInfoKHR
- Extending VkSubmitInfo:
  - VkD3D12FenceSubmitInfoKHR

New Enum Constants

- VK_KHR_EXTERNAL_SEMAPHORE_WIN32_EXTENSION_NAME
- VK_KHR_EXTERNAL_SEMAPHORE_WIN32_SPEC_VERSION
• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_D3D12_FENCE_SUBMIT_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_WIN32_HANDLE_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_KHR`

Issues

1) Do applications need to call `CloseHandle()` on the values returned from `vkGetSemaphoreWin32HandleKHR` when `handleType` is `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR`?

RESOLVED: Yes, unless it is passed back in to another driver instance to import the object. A successful get call transfers ownership of the handle to the application. Destroying the semaphore object will not destroy the handle or the handle's reference to the underlying semaphore resource.

2) Should the language regarding KMT/Windows 7 handles be moved to a separate extension so that it can be deprecated over time?

RESOLVED: No. Support for them can be deprecated by drivers if they choose, by no longer returning them in the supported handle types of the instance level queries.

3) Should applications be allowed to specify additional object attributes for shared handles?

RESOLVED: Yes. Applications will be allowed to provide similar attributes to those they would to any other handle creation API.

4) How do applications communicate the desired fence values to use with D3D12 fence-based Vulkan semaphores?

RESOLVED: There are a couple of options. The values for the signaled and reset states could be communicated up front when creating the object and remain static for the life of the Vulkan semaphore, or they could be specified using auxiliary structures when submitting semaphore signal and wait operations, similar to what is done with the keyed mutex extensions. The latter is more flexible and consistent with the keyed mutex usage, but the former is a much simpler API.

Since Vulkan tends to favor flexibility and consistency over simplicity, a new structure specifying D3D12 fence acquire and release values is added to the `vkQueueSubmit` function.

Version History

• Revision 1, 2016-10-21 (James Jones)
  ◦ Initial revision

**VK_KHR_fragment_shading_rate**

Name String

`VK_KHR_fragment_shading_rate`
Extension Type

Device extension

Registered Extension Number

227

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_create_renderpass2
• Requires VK_KHR_get_physical_device_properties2

Contact

• Tobias Hector @tobski

Other Extension Metadata

Last Modified Date

2020-05-06

Interactions and External Dependencies

• This extension requires SPV_KHR_fragment_shading_rate.

Contributors

• Tobias Hector, AMD
• Guennadi Riguer, AMD
• Matthaeus Chajdas, AMD
• Pat Brown, Nvidia
• Matthew Netsch, Qualcomm
• Slawomir Grajewski, Intel
• Jan-Harald Fredriksen, Arm
• Jeff Bolz, Nvidia
• Contributors to the VK_NV_shading_rate_image specification
• Contributors to the VK_EXT_fragment_density_map specification

Description

This extension adds the ability to change the rate at which fragments are shaded. Rather than the usual single fragment invocation for each pixel covered by a primitive, multiple pixels can be shaded by a single fragment shader invocation.

Up to three methods are available to the application to change the fragment shading rate:
• **Pipeline Fragment Shading Rate**, which allows the specification of a rate per-draw.

• **Primitive Fragment Shading Rate**, which allows the specification of a rate per primitive, specified during shading.

• **Attachment Fragment Shading Rate**, which allows the specification of a rate per-region of the framebuffer, specified in a specialized image attachment.

Additionally, these rates can all be specified and combined in order to adjust the overall detail in the image at each point.

This functionality can be used to focus shading efforts where higher levels of detail are needed in some parts of a scene compared to others. This can be particularly useful in high resolution rendering, or for XR contexts.

This extension also adds support for the `SPV_KHR_fragment_shading_rate` extension which enables setting the primitive fragment shading rate, and allows querying the final shading rate from a fragment shader.

**New Commands**

• `vkCmdSetFragmentShadingRateKHR`

• `vkGetPhysicalDeviceFragmentShadingRatesKHR`

**New Structures**

• `VkPhysicalDeviceFragmentShadingRateKHR`

• Extending `VkGraphicsPipelineCreateInfo`:
  ○ `VkPipelineFragmentShadingRateStateCreateInfoKHR`

• Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  ○ `VkPhysicalDeviceFragmentShadingRateFeaturesKHR`

• Extending `VkPhysicalDeviceProperties2`:
  ○ `VkPhysicalDeviceFragmentShadingRatePropertiesKHR`

• Extending `VkSubpassDescription2`:
  ○ `VkFragmentShadingRateAttachmentInfoKHR`

**New Enums**

• `VkFragmentShadingRateCombinerOpKHR`

**New Enum Constants**

• `VK_KHR_FRAGMENT_SHADING_RATE_EXTENSION_NAME`

• `VK_KHR_FRAGMENT_SHADING_RATE_SPEC_VERSION`

• Extending `VkAccessFlagBits`:
- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_FRAGMENT_SHADING_RATE_KHR`

- Extending `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`

- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_FRAGMENT_SHADING_RATE_ATTACHMENT_OPTIMAL_KHR`

- Extending `VkImageUsageFlagBits`:
  - `VK_IMAGE_USAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`

- Extending `VkPipelineStageFlagBits`:
  - `VK_PIPELINE_STAGE_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_FRAGMENT_SHADING_RATE_ATTACHMENT_INFO_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_PROPERTIES_KHR`
  - `VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_STATE_CREATE_INFO_KHR`

**Version History**

- Revision 1, 2020-05-06 (Tobias Hector)
  - Initial revision

**VK_KHR_get_display_properties2**

**Name String**

`VK_KHR_get_display_properties2`

**Extension Type**

Instance extension

**Registered Extension Number**

122

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_display`
Contact

• James Jones

Other Extension Metadata

Last Modified Date

2017-02-21

IP Status

No known IP claims.

Contributors

• Ian Elliott, Google
• James Jones, NVIDIA

Description

This extension provides new entry points to query device display properties and capabilities in a way that can be easily extended by other extensions, without introducing any further entry points. This extension can be considered the `VK_KHR_display` equivalent of the `VK_KHR_get_physical_device_properties2` extension.

New Commands

• `vkGetDisplayModeProperties2KHR`
• `vkGetDisplayPlaneCapabilities2KHR`
• `vkGetPhysicalDeviceDisplayPlaneProperties2KHR`
• `vkGetPhysicalDeviceDisplayProperties2KHR`

New Structures

• `VkDisplayModeProperties2KHR`
• `VkDisplayPlaneCapabilities2KHR`
• `VkDisplayPlaneInfo2KHR`
• `VkDisplayPlaneProperties2KHR`
• `VkDisplayProperties2KHR`

New Enum Constants

• `VK_KHR_GET_DISPLAY_PROPERTIES_2_EXTENSION_NAME`
• `VK_KHR_GET_DISPLAY_PROPERTIES_2_SPEC_VERSION`

Extending `VkStructureType`:

• `VK_STRUCTURE_TYPE_DISPLAY_MODE_PROPERTIES_2_KHR`
- VK_STRUCTURE_TYPE_DISPLAY_PLANE_CAPABILITIES_2_KHR
- VK_STRUCTURE_TYPE_DISPLAY_PLANE_INFO_2_KHR
- VK_STRUCTURE_TYPE_DISPLAY_PLANE_PROPERTIES_2_KHR
- VK_STRUCTURE_TYPE_DISPLAY_PROPERTIES_2_KHR

**Issues**

1) What should this extension be named?

**RESOLVED:** VK_KHR_get_display_properties2. Other alternatives:

- VK_KHR_display2
- One extension, combined with VK_KHR_surface_capabilities2.

2) Should extensible input structs be added for these new functions:

**RESOLVED:**

- vkGetPhysicalDeviceDisplayProperties2KHR: No. The only current input is a VkPhysicalDevice. Other inputs wouldn't make sense.
- vkGetPhysicalDeviceDisplayPlaneProperties2KHR: No. The only current input is a VkPhysicalDevice. Other inputs wouldn't make sense.
- vkGetDisplayModeProperties2KHR: No. The only current inputs are a VkPhysicalDevice and a VkDisplayModeKHR. Other inputs wouldn't make sense.

3) Should additional display query functions be extended?

**RESOLVED:**

- vkGetDisplayPlaneSupportedDisplaysKHR: No. Extensions should instead extend vkGetDisplayPlaneCapabilitiesKHR().

**Version History**

- Revision 1, 2017-02-21 (James Jones)
  - Initial draft.

**VK_KHR_get_surface_capabilities2**

**Name String**

VK_KHR_get_surface_capabilities2

**Extension Type**

Instance extension

**Registered Extension Number**

120
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

Contact

- James Jones 📞cubanismo

Other Extension Metadata

Last Modified Date

2017-02-27

IP Status

No known IP claims.

Contributors

- Ian Elliott, Google
- James Jones, NVIDIA
- Alon Or-bach, Samsung

Description

This extension provides new entry points to query device surface capabilities in a way that can be easily extended by other extensions, without introducing any further entry points. This extension can be considered the `VK_KHR_surface` equivalent of the `VK_KHR_get_physical_device_properties2` extension.

New Commands

- `vkGetPhysicalDeviceSurfaceCapabilities2KHR`
- `vkGetPhysicalDeviceSurfaceFormats2KHR`

New Structures

- `VkPhysicalDeviceSurfaceInfo2KHR`
- `VkSurfaceCapabilities2KHR`
- `VkSurfaceFormat2KHR`

New Enum Constants

- `VK_KHR_GET_SURFACE_CAPABILITIES_2_EXTENSION_NAME`
- `VK_KHR_GET_SURFACE_CAPABILITIES_2_SPEC_VERSION`
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SURFACE_INFO_2_KHR
  ◦ VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_KHR
  ◦ VK_STRUCTURE_TYPE_SURFACE_FORMAT_2_KHR

Issues

1) What should this extension be named?

RESOLVED: VK_KHR_get_surface_capabilities2. Other alternatives:

• VK_KHR_surface2

• One extension, combining a separate display-specific query extension.

2) Should additional WSI query functions be extended?

RESOLVED:

• vkGetPhysicalDeviceSurfaceCapabilitiesKHR: Yes. The need for this motivated the extension.
• vkGetPhysicalDeviceSurfaceSupportKHR: No. Currently only has boolean output. Extensions should instead extend vkGetPhysicalDeviceSurfaceCapabilities2KHR.
• vkGetPhysicalDeviceSurfaceFormatsKHR: Yes.
• vkGetPhysicalDeviceSurfacePresentModesKHR: No. Recent discussion concluded this introduced too much variability for applications to deal with. Extensions should instead extend vkGetPhysicalDeviceSurfaceCapabilities2KHR.
• vkGetPhysicalDeviceXlibPresentationSupportKHR: Not in this extension.
• vkGetPhysicalDeviceXcbPresentationSupportKHR: Not in this extension.
• vkGetPhysicalDeviceWaylandPresentationSupportKHR: Not in this extension.
• vkGetPhysicalDeviceWin32PresentationSupportKHR: Not in this extension.

Version History

• Revision 1, 2017-02-27 (James Jones)
  ◦ Initial draft.

VK_KHR_image_format_list

Name String
  VK_KHR_image_format_list

Extension Type
  Device extension
On some implementations, setting the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` on image creation can cause access to that image to perform worse than an equivalent image created without `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` because the implementation does not know what view formats will be paired with the image.

This extension allows an application to provide the list of all formats that can be used with an image when it is created. The implementation may then be able to create a more efficient image that supports the subset of formats required by the application without having to support all formats in the format compatibility class of the image format.
Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending VkImageCreateInfo, VkSwapchainCreateInfoKHR, VkPhysicalDeviceImageFormatInfo2:
  - VkImageFormatListCreateInfoKHR

New Enum Constants

- VK_KHR_IMAGE_FORMAT_LIST_EXTENSION_NAME
- VK_KHR_IMAGE_FORMAT_LIST_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMAGE_FORMAT_LIST_CREATE_INFO_KHR

Version History

- Revision 1, 2017-03-20 (Jason Ekstrand)
  - Initial revision

VK_KHR_imageless_framebuffer

Name String

VK_KHR_imageless_framebuffer

Extension Type

Device extension

Registered Extension Number

109

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_maintenance2
- Requires VK_KHR_image_format_list

Deprecation state

- Promoted to Vulkan 1.2
Description

This extension allows framebuffers to be created without the need for creating images first, allowing more flexibility in how they are used, and avoiding the need for many of the confusing compatibility rules.

Framebuffers are now created with a small amount of additional metadata about the image views that will be used in `VkFramebufferAttachmentsCreateInfoKHR`, and the actual image views are provided at render pass begin time via `VkRenderPassAttachmentBeginInfoKHR`.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- `VkFramebufferAttachmentImageInfoKHR`
- Extending `VkFramebufferCreateInfo`:
  - `VkFramebufferAttachmentsCreateInfoKHR`
- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceImagelessFramebufferFeaturesKHR`
- Extending `VkRenderPassBeginInfo`:
  - `VkRenderPassAttachmentBeginInfoKHR`

New Enum Constants

- `VK_KHR_IMAGELESS_FRAMEBUFFER_EXTENSION_NAME`
- `VK_KHR_IMAGELESS_FRAMEBUFFER_SPEC_VERSION`
- Extending `VkFramebufferCreateFlagBits`:
Extending `VkStructureType`:
- `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENTS_CREATE_INFO_KHR`
- `VK_STRUCTURE_TYPE_FRAMEBUFFER_ATTACHMENT_IMAGE_INFO_KHR`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGELESS_FRAMEBUFFER_FEATURES_KHR`
- `VK_STRUCTURE_TYPE_RENDER_PASS_ATTACHMENT_BEGIN_INFO_KHR`

Version History

- Revision 1, 2018-12-14 (Tobias Hector)
  - Internal revisions

**VK_KHR_incremental_present**

Name String

`VK_KHR_incremental_present`

Extension Type

Device extension

Registered Extension Number

85

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_swapchain`

Contact

- Ian Elliott @ianelliottus

Other Extension Metadata

Last Modified Date

2016-11-02

IP Status

No known IP claims.

Contributors

- Ian Elliott, Google
- Jesse Hall, Google
Description

This device extension extends `vkQueuePresentKHR` from the `VK_KHR_swapchain` extension, allowing an application to specify a list of rectangular, modified regions of each image to present. This should be used in situations where an application is only changing a small portion of the presentable images within a swapchain, since it enables the presentation engine to avoid wasting time presenting parts of the surface that have not changed.

This extension is leveraged from the `EGL_KHR_swap_buffers_with_damage` extension.

New Structures

- `VkPresentRegionKHR`
- `VkRectLayerKHR`
- Extending `VkPresentInfoKHR`:
  - `VkPresentRegionsKHR`

New Enum Constants

- `VK_KHR_INCREMENTAL_PRESENT_EXTENSION_NAME`
- `VK_KHR_INCREMENTAL_PRESENT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PRESENT_REGIONS_KHR`

Issues

1) How should we handle steroescopic-3D swapchains? We need to add a layer for each rectangle. One approach is to create another struct containing the `VkRect2D` plus layer, and have `VkPresentRegionsKHR` point to an array of that struct. Another approach is to have two parallel arrays, `pRectangles` and `pLayers`, where `pRectangles[i]` and `pLayers[i]` must be used together. Which approach should we use, and if the array of a new structure, what should that be called?

**RESOLVED**: Create a new structure, which is a `VkRect2D` plus a layer, and will be called `VkRectLayerKHR`.

2) Where is the origin of the `VkRectLayerKHR`?
RESOLVED: The upper left corner of the presentable image(s) of the swapchain, per the definition of framebuffer coordinates.

3) Does the rectangular region, VkRectLayerKHR, specify pixels of the swapchain’s image(s), or of the surface?

RESOLVED: Of the image(s). Some presentation engines may scale the pixels of a swapchain’s image(s) to the size of the surface. The size of the swapchain’s image(s) will be consistent, where the size of the surface may vary over time.

4) What if all of the rectangles for a given swapchain contain a width and/or height of zero?

RESOLVED: The application is indicating that no pixels changed since the last present. The presentation engine may use such a hint and not update any pixels for the swapchain. However, all other semantics of vkQueuePresentKHR must still be honored, including waiting for semaphores to signal.

5) When the swapchain is created with VkSwapchainCreateInfoKHR::preTransform set to a value other than VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR, should the rectangular region, VkRectLayerKHR, be transformed to align with the preTransform?

RESOLVED: No. The rectangular region in VkRectLayerKHR should not be tranformed. As such, it may not align with the extents of the swapchain’s image(s). It is the responsibility of the presentation engine to transform the rectangular region. This matches the behavior of the Android presentation engine, which set the precedent.

Version History

- Revision 1, 2016-11-02 (Ian Elliott)
  - Internal revisions
- Revision 2, 2021-03-18 (Ian Elliott)
  - Clarified alignment of rectangles for presentation engines that support transformed swapchains.

VK_KHR_performance_query

Name String

VK_KHR_performance_query

Extension Type

Device extension

Registered Extension Number

117

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Special Use

- Developer tools

Contact

- Alon Or-bach (@alonorbach)

Other Extension Metadata

Last Modified Date

2019-10-08

IP Status

No known IP claims.

Contributors

- Jesse Barker, Unity Technologies
- Kenneth Benzie, Codeplay
- Jan-Harald Fredriksen, ARM
- Jeff Leger, Qualcomm
- Jesse Hall, Google
- Tobias Hector, AMD
- Neil Henning, Codeplay
- Baldur Karlsson
- Lionel Landwerlin, Intel
- Peter Lohrmann, AMD
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Niklas Smedberg, Unity Technologies
- Igor Ostrowski, Intel

Description

The VK_KHR_performance_query extension adds a mechanism to allow querying of performance counters for use in applications and by profiling tools.

Each queue family may expose counters that can be enabled on a queue of that family. We extend VkQueryType to add a new query type for performance queries, and chain a structure on VkQueryPoolCreateInfo to specify the performance queries to enable.
New Commands

- `vkAcquireProfilingLockKHR`
- `vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR`
- `vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR`
- `vkReleaseProfilingLockKHR`

New Structures

- `VkAcquireProfilingLockInfoKHR`
- `VkPerformanceCounterDescriptionKHR`
- `VkPerformanceCounterKHR`
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDevicePerformanceQueryFeaturesKHR`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDevicePerformanceQueryPropertiesKHR`
- Extending `VkQueryPoolCreateInfo`:
  - `VkQueryPoolPerformanceCreateInfoKHR`
- Extending `VkSubmitInfo`, `VkSubmitInfo2KHR`:
  - `VkPerformanceQuerySubmitInfoKHR`

New Unions

- `VkPerformanceCounterResultKHR`

New Enums

- `VkAcquireProfilingLockFlagBitsKHR`
- `VkPerformanceCounterDescriptionFlagBitsKHR`
- `VkPerformanceCounterScopeKHR`
- `VkPerformanceCounterStorageKHR`
- `VkPerformanceCounterUnitKHR`

New Bitmasks

- `VkAcquireProfilingLockFlagsKHR`
- `VkPerformanceCounterDescriptionFlagsKHR`

New Enum Constants

- `VK_KHR_PERFORMANCE_QUERY_EXTENSION_NAME`
• VK_KHR_PERFORMANCE_QUERY_SPEC_VERSION

Extending VkQueryType:
  ◦ VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR

Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_DESCRIPTION_KHR
  ◦ VK_STRUCTURE_TYPE_PERFORMANCE_COUNTER_KHR
  ◦ VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_FEATURES_KHR
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PERFORMANCE_QUERY_PROPERTIES_KHR
  ◦ VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR

Issues

1) Should this extension include a mechanism to begin a query in command buffer A and end the query in command buffer B?

RESOLVED No - queries are tied to command buffer creation and thus have to be encapsulated within a single command buffer.

2) Should this extension include a mechanism to begin and end queries globally on the queue, not using the existing command buffer commands?

RESOLVED No - for the same reasoning as the resolution of 1).

3) Should this extension expose counters that require multiple passes?

RESOLVED Yes - users should re-submit a command buffer with the same commands in it multiple times, specifying the pass to count as the query parameter in VkPerformanceQuerySubmitInfoKHR.

4) How to handle counters across parallel workloads?

RESOLVED In the spirit of Vulkan, a counter description flag VK_PERFORMANCE_COUNTER_DESCRIPTION_CONCURRENTLY_IMPACTED_BIT_KHR denotes that the accuracy of a counter result is affected by parallel workloads.

5) How to handle secondary command buffers?

RESOLVED Secondary command buffers inherit any counter pass index specified in the parent primary command buffer. Note: this is no longer an issue after change from issue 10 resolution.

6) What commands does the profiling lock have to be held for?

RESOLVED For any command buffer that is being queried with a performance query pool, the profiling lock must be held while that command buffer is in the recording, executable, or pending state.
7) Should we support `vkCmdCopyQueryPoolResults`?

**RESOLVED** Yes.

8) Should we allow performance queries to interact with multiview?

**RESOLVED** Yes, but the performance queries must be performed once for each pass per view.

9) Should a `queryCount > 1` be usable for performance queries?

**RESOLVED** Yes. Some vendors will have costly performance counter query pool creation, and would rather if a certain set of counters were to be used multiple times that a `queryCount > 1` can be used to amortize the instantiation cost.

10) Should we introduce an indirect mechanism to set the counter pass index?

**RESOLVED** Specify the counter pass index at submit time instead, to avoid requiring re-recording of command buffers when multiple counter passes are needed.

**Examples**

The following example shows how to find what performance counters a queue family supports, setup a query pool to record these performance counters, how to add the query pool to the command buffer to record information, and how to get the results from the query pool.

```cpp
// A previously created physical device
VkPhysicalDevice physicalDevice;

// One of the queue families our device supports
uint32_t queueFamilyIndex;

uint32_t counterCount;

// Get the count of counters supported
vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(physicalDevice,
queueFamilyIndex,
&counterCount,
NULL,
NULL);

VkPerformanceCounterKHR* counters = malloc(sizeof(VkPerformanceCounterKHR) * counterCount);
VkPerformanceCounterDescriptionKHR* counterDescriptions = malloc(sizeof(VkPerformanceCounterDescriptionKHR) * counterCount);

// Get the counters supported
vkEnumeratePhysicalDeviceQueueFamilyPerformanceQueryCountersKHR(physicalDevice,
queueFamilyIndex,
&counterCount,

// ...
counters,
    counterDescriptions);

// Try to enable the first 8 counters
uint32_t enabledCounters[8];

const uint32_t enabledCounterCount = min(counterCount, 8));

for (uint32_t i = 0; i < enabledCounterCount; i++) {
    enabledCounters[i] = i;
}

// A previously created device that had the performanceCounterQueryPools feature
// set to VK_TRUE
VkDevice device;

VkQueryPoolPerformanceCreateInfoKHR performanceQueryCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_CREATE_INFO_KHR,
    NULL,
    // Specify the queue family that this performance query is performed on
    queueFamilyIndex,
    // The number of counters to enable
    enabledCounterCount,
    // The array of indices of counters to enable
    enabledCounters
};

// Get the number of passes our counters will require.
uint32_t numPasses;

vkGetPhysicalDeviceQueueFamilyPerformanceQueryPassesKHR(
    physicalDevice,
    &performanceQueryCreateInfo,
    &numPasses);

VkQueryPoolCreateInfo queryPoolCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO,
    &performanceQueryCreateInfo,
    0,
    // Using our new query type here
    VK_QUERY_TYPE_PERFORMANCE_QUERY_KHR,
    1,
    0
};

Appendix E: Layers & Extensions (Informative) | 2577
VkQueryPool queryPool;

VkResult result = vkCreateQueryPool(
    device,
    &queryPoolCreateInfo,
    NULL,
    &queryPool);

assert(VK_SUCCESS == result);

// A queue from queueFamilyIndex
VkQueue queue;

// A command buffer we want to record counters on
VkCommandBuffer commandBuffer;

VkCommandBufferBeginInfo commandBufferBeginInfo = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    NULL,
    0,
    NULL
};

VkAcquireProfilingLockInfoKHR lockInfo = {
    VK_STRUCTURE_TYPE_ACQUIRE_PROFILING_LOCK_INFO_KHR,
    NULL,
    0,
    UINT64_MAX // Wait forever for the lock
};

// Acquire the profiling lock before we record command buffers
// that will use performance queries

result = vkAcquireProfilingLockKHR(device, &lockInfo);

assert(VK_SUCCESS == result);

result = vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);

assert(VK_SUCCESS == result);

vkCmdResetQueryPool(
    commandBuffer,
    queryPool,
    0,
    1);

vkCmdBeginQuery(
    commandBuffer,
    queryPool,
// Perform the commands you want to get performance information on
// ...

// Perform a barrier to ensure all previous commands were complete before
// ending the query
vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
    0,
    0,
    NULL,
    0,
    NULL,
    0,
    NULL);

vkCmdEndQuery(
    commandBuffer,
    queryPool,
    0);

result = vkEndCommandBuffer(commandBuffer);
assert(VK_SUCCESS == result);

for (uint32_t counterPass = 0; counterPass < numPasses; counterPass++) {

    VkPerformanceQuerySubmitInfoKHR performanceQuerySubmitInfo = {
        VK_STRUCTURE_TYPE_PERFORMANCE_QUERY_SUBMIT_INFO_KHR,
        NULL,
        counterPass
    };

    // Submit the command buffer and wait for its completion
    // ...
}

// Release the profiling lock after the command buffer is no longer in the
// pending state.
vkReleaseProfilingLockKHR(device);

result = vkResetCommandBuffer(commandBuffer, 0);
assert(VK_SUCCESS == result);

// Create an array to hold the results of all counters
VkPerformanceCounterResultKHR* recordedCounters = malloc(0, 0);

Appendix E: Layers & Extensions (Informative) | 2579
result = vkGetQueryPoolResults(
    device,
    queryPool,
    0,
    1,
    sizeof(VkPerformanceCounterResultKHR) * enabledCounterCount,
    recordedCounters,
    sizeof(VkPerformanceCounterResultKHR),
    NULL);

// recordedCounters is filled with our counters, we'll look at one for posterity
switch (counters[0].storage) {
    case VK_PERFORMANCE_COUNTER_STORAGE_INT32:
        // use recordCounters[0].int32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_INT64:
        // use recordCounters[0].int64 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_UINT32:
        // use recordCounters[0].uint32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_UINT64:
        // use recordCounters[0].uint64 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_FLOAT32:
        // use recordCounters[0].float32 to get at the counter result!
        break;
    case VK_PERFORMANCE_COUNTER_STORAGE_FLOAT64:
        // use recordCounters[0].float64 to get at the counter result!
        break;
}

Version History

• Revision 1, 2019-10-08

VK_KHR_pipeline_executable_properties

Name String

VK_KHR_pipeline_executable_properties

Extension Type

Device extension

Registered Extension Number

270
Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Special Use

• Developer tools

Contact

• Jason Ekstrand @jekstrand

Other Extension Metadata

Last Modified Date

2019-05-28

IP Status

No known IP claims.

Interactions and External Dependencies

Contributors

• Jason Ekstrand, Intel
• Ian Romanick, Intel
• Kenneth Graunke, Intel
• Baldur Karlsson, Valve
• Jesse Hall, Google
• Jeff Bolz, Nvidia
• Piers Daniel, Nvidia
• Tobias Hector, AMD
• Jan-Harald Fredriksen, ARM
• Tom Olson, ARM
• Daniel Koch, Nvidia
• Spencer Fricke, Samsung

Description

When a pipeline is created, its state and shaders are compiled into zero or more device-specific executables, which are used when executing commands against that pipeline. This extension adds a mechanism to query properties and statistics about the different executables produced by the pipeline compilation process. This is intended to be used by debugging and performance tools to allow them to provide more detailed information to the user. Certain compile-time shader statistics...
provided through this extension may be useful to developers for debugging or performance analysis.

**New Commands**

- vkGetPipelineExecutableInternalRepresentationsKHR
- vkGetPipelineExecutablePropertiesKHR
- vkGetPipelineExecutableStatisticsKHR

**New Structures**

- VkPipelineExecutableInfoKHR
- VkPipelineExecutableInternalRepresentationKHR
- VkPipelineExecutablePropertiesKHR
- VkPipelineExecutableStatisticKHR
- VkPipelineInfoKHR
  - Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
    - VkPhysicalDevicePipelineExecutablePropertiesFeaturesKHR

**New Unions**

- VkPipelineExecutableStatisticValueKHR

**New Enums**

- VkPipelineExecutableStatisticFormatKHR

**New Enum Constants**

- VK_KHR_PIPELINE_EXECUTABLE_PROPERTIES_EXTENSION_NAME
- VK_KHR_PIPELINE_EXECUTABLE_PROPERTIES_SPEC_VERSION
  - Extending VkPipelineCreateFlagBits:
    - VK_PIPELINE_CREATE_CAPTURE_INTERNAL_REPRESENTATIONS_BIT_KHR
    - VK_PIPELINE_CREATE_CAPTURE_STATISTICS_BIT_KHR
  - Extending VkStructureType:
    - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_EXECUTABLE_PROPERTIES_FEATURES_KHR
    - VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INFO_KHR
    - VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_INTERNAL_REPRESENTATION_KHR
    - VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_PROPERTIES_KHR
    - VK_STRUCTURE_TYPE_PIPELINE_EXECUTABLE_STATISTIC_KHR
    - VK_STRUCTURE_TYPE_PIPELINE_INFO_KHR

2582 | Appendix E: Layers & Extensions (Informative)
**Issues**

1) What should we call the pieces of the pipeline which are produced by the compilation process and about which you can query properties and statistics?

**RESOLVED:** Call them “executables”. The name “binary” was used in early drafts of the extension but it was determined that “pipeline binary” could have a fairly broad meaning (such as a binary serialized form of an entire pipeline) and was too big of a namespace for the very specific needs of this extension.

**Version History**

- Revision 1, 2019-05-28 (Jason Ekstrand)
  - Initial draft

**VK_KHR_pipeline_library**

**Name String**

`VK_KHR_pipeline_library`

**Extension Type**

Device extension

**Registered Extension Number**

291

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Christoph Kubisch [pixeljetstream](https://pixeljetstream)

**Other Extension Metadata**

**Last Modified Date**

2020-01-08

**IP Status**

No known IP claims.

**Contributors**

- See contributors to `VK_KHR_ray_tracing`
Description

A pipeline library is a special pipeline that cannot be bound, instead it defines a set of shaders and shader groups which can be linked into other pipelines. This extension defines the infrastructure for pipeline libraries, but does not specify the creation or usage of pipeline libraries. This is left to additional dependent extensions.

New Structures

- VkPipelineLibraryCreateInfoKHR

New Enum Constants

- VK_KHR_PIPELINE_LIBRARY_EXTENSION_NAME
- VK_KHR_PIPELINE_LIBRARY_SPEC_VERSION
- Extending VkPipelineCreateFlagBits:
  - VK_PIPELINE_CREATE_LIBRARY_BIT_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PIPELINE_LIBRARY_CREATE_INFO_KHR

Version History

- Revision 1, 2020-01-08 (Christoph Kubisch)
  - Initial draft.

VK_KHR_push_descriptor

Name String

VK_KHR_push_descriptor

Extension Type

Device extension

Registered Extension Number

81

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Jeff Bolz @jeffbolznv
Other Extension Metadata

Last Modified Date
2017-09-12

IP Status
No known IP claims.

Contributors
• Jeff Bolz, NVIDIA
• Michael Worcester, Imagination Technologies

Description
This extension allows descriptors to be written into the command buffer, while the implementation is responsible for managing their memory. Push descriptors may enable easier porting from older APIs and in some cases can be more efficient than writing descriptors into descriptor sets.

New Commands

• `vkCmdPushDescriptorSetKHR`

If `VK_KHR_descriptor_update_template` is supported:

• `vkCmdPushDescriptorSetWithTemplateKHR`

If Version 1.1 is supported:

• `vkCmdPushDescriptorSetWithTemplateKHR`

New Structures

• Extending `VkPhysicalDeviceProperties2`:
  ◦ `VkPhysicalDevicePushDescriptorPropertiesKHR`

New Enum Constants

• `VK_KHR_PUSH_DESCRIPTOR_EXTENSION_NAME`
• `VK_KHR_PUSH_DESCRIPTOR_SPEC_VERSION`

• Extending `VkDescriptorSetLayoutCreateFlagBits`:
  ◦ `VK_DESCRIPTOR_SET_LAYOUT_CREATE_PUSH_DESCRIPTOR_BIT_KHR`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PUSH_DESCRIPTOR_PROPERTIES_KHR`

If `VK_KHR_descriptor_update_template` is supported:

• Extending `VkDescriptorUpdateTemplateType`:

Appendix E: Layers & Extensions (Informative) | 2585
If **Version 1.1** is supported:

- Extending `VkDescriptorUpdateTemplateType`:
  - `VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPERTORS_KHR`

**Version History**

- Revision 1, 2016-10-15 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-09-12 (Tobias Hector)
  - Added interactions with Vulkan 1.1

**VK_KHR_ray_query**

**Name String**

- `VK_KHR_ray_query`

**Extension Type**

- Device extension

**Registered Extension Number**

- 349

**Revision**

- 1

**Extension and Version Dependencies**

- Requires Vulkan 1.1
- Requires `VK_KHR_spirv_1_4`
- Requires `VK_KHR_acceleration_structure`

**Contact**

- Daniel Koch [dgkoch](mailto:dgkoch)

**Other Extension Metadata**

**Last Modified Date**

- 2020-11-12

**Interactions and External Dependencies**

- This extension requires `SPV_KHR_ray_query`
- This extension provides API support for `GLSL_EXT_ray_query`
Contributors

- Matthäus Chajdas, AMD
- Greg Grebe, AMD
- Nicolai Hähnle, AMD
- Tobias Hector, AMD
- Dave Oldcorn, AMD
- Skyler Saleh, AMD
- Mathieu Robart, Arm
- Marius Bjorge, Arm
- Tom Olson, Arm
- Sebastian Tafuri, EA
- Henrik Rydgard, Embark
- Juan Cañada, Epic Games
- Patrick Kelly, Epic Games
- Yuriy O’Donnell, Epic Games
- Michael Doggett, Facebook/Oculus
- Andrew Garrard, Imagination
- Don Scorgie, Imagination
- Dae Kim, Imagination
- Joshua Barczak, Intel
- Slawek Grajewski, Intel
- Jeff Bolz, NVIDIA
- Pascal Gautron, NVIDIA
- Daniel Koch, NVIDIA
- Christoph Kubisch, NVIDIA
- Ashwin Lele, NVIDIA
- Robert Stepinski, NVIDIA
- Martin Stich, NVIDIA
- Nuno Subtil, NVIDIA
- Eric Werness, NVIDIA
- Jon Leech, Khronos
- Jeroen van Schijndel, OTOY
- Juul Joosten, OTOY
- Alex Bourd, Qualcomm
- Roman Larionov, Qualcomm
Description

Rasterization has been the dominant method to produce interactive graphics, but increasing performance of graphics hardware has made ray tracing a viable option for interactive rendering. Being able to integrate ray tracing with traditional rasterization makes it easier for applications to incrementally add ray traced effects to existing applications or to do hybrid approaches with rasterization for primary visibility and ray tracing for secondary queries.

Ray queries are available to all shader types, including graphics, compute and ray tracing pipelines. Ray queries are not able to launch additional shaders, instead returning traversal results to the calling shader.

This extension adds support for the following SPIR-V extension in Vulkan:

- **SPV_KHR_ray_query**

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceRayQueryFeaturesKHR`

New Enum Constants

- `VK_KHR_RAY_QUERY_EXTENSION_NAME`
- `VK_KHR_RAY_QUERY_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_QUERY_FEATURES_KHR`

New SPIR-V Capabilities

- `RayQueryKHR`
- `RayTraversalPrimitiveCullingKHR`

Sample Code

Example of ray query in a GLSL shader
Issues

(1) What are the changes between the public provisional (VK_KHR_ray_tracing v8) release and the final (VK_KHR_acceleration_structure v11 / VK_KHR_ray_query v1) release?

- refactor VK_KHR_ray_tracing into 3 extensions, enabling implementation flexibility and decoupling ray query support from ray pipelines:
  - VK_KHR_acceleration_structure (for acceleration structure operations)
  - VK_KHR_ray_tracing_pipeline (for ray tracing pipeline and shader stages)
  - VK_KHR_ray_query (for ray queries in existing shader stages)
- Update SPIRV capabilities to use RayQueryKHR
- extension is no longer provisional

Version History

- Revision 1, 2020-11-12 (Mathieu Robart, Daniel Koch, Andrew Garrard)
  - Decomposition of the specification, from VK_KHR_ray_tracing to VK_KHR_ray_query (#1918,!3912)
  - update to use RayQueryKHR SPIR-V capability
  - add numerical limits for ray parameters (#2235,!3960)
  - relax formula for ray intersection candidate determination (#2322,!4080)
  - restrict traces to TLAS (#2239,!4141)
  - require HitT to be in ray interval for OpRayQueryGenerateIntersectionKHR (#2359,!4146)
  - add ray query shader stages for AS read bit (#2407,!4203)
VK_KHR_ray_tracing_pipeline

Name String

VK_KHR_ray_tracing_pipeline

Extension Type

Device extension

Registered Extension Number

348

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.1
• Requires VK_KHR_spirv_1_4
• Requires VK_KHR_acceleration_structure

Contact

• Daniel Koch @dgkoch

Other Extension Metadata

Last Modified Date

2020-11-12

Interactions and External Dependencies

• This extension requires SPV_KHR_ray_tracing
• This extension provides API support for GLSL_EXT_ray_tracing
• This extension interacts with Vulkan 1.2 and VK_KHR_vulkan_memory_model, adding the shader-call-related relation of invocations, shader-call-order partial order of dynamic instances of instructions, and the ShaderCallKHR scope.
• This extension interacts with VK_KHR_pipeline_library, enabling pipeline libraries to be used with ray tracing pipelines and enabling usage of VkRayTracingPipelineInterfaceCreateInfoKHR.

Contributors

• Matthäus Chajdas, AMD
• Greg Grebe, AMD
• Nicolai Hähnle, AMD
• Tobias Hector, AMD
• Dave Oldcorn, AMD
• Skyler Saleh, AMD
• Mathieu Robart, Arm
• Marius Bjorge, Arm
• Tom Olson, Arm
• Sebastian Tafuri, EA
• Henrik Rydgard, Embark
• Juan Cañada, Epic Games
• Patrick Kelly, Epic Games
• Yuriy O’Donnell, Epic Games
• Michael Doggett, Facebook/Oculus
• Andrew Garrard, Imagination
• Don Scorgie, Imagination
• Dae Kim, Imagination
• Joshua Barczak, Intel
• Slawek Grajewski, Intel
• Jeff Bolz, NVIDIA
• Pascal Gautron, NVIDIA
• Daniel Koch, NVIDIA
• Christoph Kubisch, NVIDIA
• Ashwin Lele, NVIDIA
• Robert Stepinski, NVIDIA
• Martin Stich, NVIDIA
• Nuno Subtil, NVIDIA
• Eric Werness, NVIDIA
• Jon Leech, Khronos
• Jeroen van Schijndel, OTOY
• Juul Joosten, OTOY
• Alex Bourd, Qualcomm
• Roman Larionov, Qualcomm
• David McAllister, Qualcomm
• Spencer Fricke, Samsung
• Lewis Gordon, Samsung
• Ralph Potter, Samsung
• Jasper Bekkers, Traverse Research
• Jesse Barker, Unity
• Baldur Karlsson, Valve
Description

Rasterization has been the dominant method to produce interactive graphics, but increasing performance of graphics hardware has made ray tracing a viable option for interactive rendering. Being able to integrate ray tracing with traditional rasterization makes it easier for applications to incrementally add ray traced effects to existing applications or to do hybrid approaches with rasterization for primary visibility and ray tracing for secondary queries.

To enable ray tracing, this extension adds a few different categories of new functionality:

- A new ray tracing pipeline type with new shader domains: ray generation, intersection, any-hit, closest hit, miss, and callable
- A shader binding indirection table to link shader groups with acceleration structure items
- Trace ray commands which initiates the ray pipeline traversal and invocation of the various new shader domains depending on which traversal conditions are met

This extension adds support for the following SPIR-V extension in Vulkan:

- SPV_KHR_ray_tracing

New Commands

- vkCmdSetRayTracingPipelineStackSizeKHR
- vkCmdTraceRaysIndirectKHR
- vkCmdTraceRaysKHR
- vkCreateRayTracingPipelinesKHR
- vkGetRayTracingCaptureReplayShaderGroupHandlesKHR
- vkGetRayTracingShaderGroupHandlesKHR
- vkGetRayTracingShaderGroupStackSizeKHR

New Structures

- VkRayTracingPipelineCreateInfoKHR
- VkRayTracingPipelineInterfaceCreateInfoKHR
- VkRayTracingShaderGroupCreateInfoKHR
- VkStridedDeviceAddressRegionKHR
- VkTraceRaysIndirectCommandKHR

Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:

- VkPhysicalDeviceRayTracingPipelineFeaturesKHR

Extending VkPhysicalDeviceProperties2:

- VkPhysicalDeviceRayTracingPipelinePropertiesKHR
New Enums

- VkRayTracingShaderGroupTypeKHR
- VkShaderGroupShaderKHR

New Enum Constants

- VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME
- VK_KHR_RAY_TRACING_PIPELINE_SPEC_VERSION
- VK_SHADER_UNUSED_KHR

Extending VkBufferUsageFlagBits:
- VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR

Extending VkDynamicState:
- VK_DYNAMIC_STATE_RAY_TRACING_PIPELINE_STACK_SIZE_KHR

Extending VkPipelineBindPoint:
- VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR

Extending VkPipelineCreateFlagBits:
- VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ANY_HIT_SHADERS_BIT_KHR
- VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_CLOSEST_HIT_SHADERS_BIT_KHR
- VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_INTERSECTION_SHADERS_BIT_KHR
- VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_MISS_SHADERS_BIT_KHR
- VK_PIPELINE_CREATE_RAY_TRACING_SHADER_GROUP_HANDLE_CAPTURE_REPLAY_BIT_KHR
- VK_PIPELINE_CREATE_RAY_TRACING_SKIP_AABBS_BIT_KHR
- VK_PIPELINE_CREATE_RAY_TRACING_SKIP_TRIANGLES_BIT_KHR

Extending VkPipelineStageFlagBits:
- VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR

Extending VkShaderStageFlagBits:
- VK_SHADER_STAGE_ANY_HIT_BIT_KHR
- VK_SHADER_STAGE_CALLABLE_BIT_KHR
- VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR
- VK_SHADER_STAGE_INTERSECTION_BIT_KHR
- VK_SHADER_STAGE_MISS_BIT_KHR
- VK_SHADER_STAGE_RAYGEN_BIT_KHR

Extending VkStructureType:
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_PROPERTIES_KHR
New or Modified Built-In Variables

- LaunchIdKHR
- LaunchSizeKHR
- WorldRayOriginKHR
- WorldRayDirectionKHR
- ObjectRayOriginKHR
- ObjectRayDirectionKHR
- RayTminKHR
- RayTmaxKHR
- InstanceCustomIndexKHR
- InstanceId
- ObjectToWorldKHR
- WorldToObjectKHR
- HitKindKHR
- IncomingRayFlagsKHR
- RayGeometryIndexKHR
- (modified) PrimitiveId

New SPIR-V Capabilities

- RayTracingKHR
- RayTraversalPrimitiveCullingKHR

Issues

(1) How does this extension differ from VK_NV_ray_tracing?

DISCUSSION:

The following is a summary of the main functional differences between VK_KHR_ray_tracing_pipeline and VK_NV_ray_tracing:

- added support for indirect ray tracing (vkCmdTraceRaysIndirectKHR)
- uses SPV_KHR_ray_tracing instead of SPV_NV_ray_tracing
  - refer to KHR SPIR-V enums instead of NV SPIR-V enums (which are functionally equivalent and aliased to the same values).
• added RayGeometryIndexKHR built-in

• removed vkCompileDeferredNV compilation functionality and replaced with deferred host operations interactions for ray tracing

• added VkPhysicalDeviceRayTracingPipelineFeaturesKHR structure

• extended VkPhysicalDeviceRayTracingPipelinePropertiesKHR structure
  ◦ renamed maxRecursionDepth to maxRayRecursionDepth and it has a minimum of 1 instead of 31
  ◦ require shaderGroupHandleSize to be 32 bytes
  ◦ added maxRayDispatchInvocationCount, shaderGroupHandleAlignment and maxRayHitAttributeSize

• reworked geometry structures so they could be better shared between device, host, and indirect builds

• changed SBT parameters to a structure and added size (VkStridedDeviceAddressRegionKHR)

• add parameter for requesting memory requirements for host and/or device build

• added pipeline library support for ray tracing

• added watertightness guarantees

• added no-null-shader pipeline flags (VK_PIPELINE_CREATE_RAY_TRACING_NO_NULL_ *_SHADERS_BIT_KHR)

• added memory model interactions with ray tracing and define how subgroups work and can be repacked

(2) Can you give a more detailed comparison of differences and similarities between VK_NV_ray_tracing and VK_KHR_ray_tracing_pipeline?

DISCUSSION:

The following is a more detailed comparison of which commands, structures, and enums are aliased, changed, or removed.

• Aliased functionality — enums, structures, and commands that are considered equivalent:
  ◦ VkRayTracingShaderGroupTypeNV ↔ VkRayTracingShaderGroupTypeKHR
  ◦ vkGetRayTracingShaderGroupHandlesNV ↔ vkGetRayTracingShaderGroupHandlesKHR

• Changed enums, structures, and commands:
  ◦ VkRayTracingShaderGroupCreateInfoNV → VkRayTracingShaderGroupCreateInfoKHR
    (added pShaderGroupCaptureReplayHandle)
  ◦ VkRayTracingPipelineCreateInfoNV → VkRayTracingPipelineCreateInfoKHR (changed type of pGroups, added libraries, pLibraryInterface, and pDynamicState)
  ◦ VkPhysicalDeviceRayTracingPropertiesNV → VkPhysicalDeviceRayTracingPropertiesKHR
    (renamed maxTriangleCount to maxPrimitiveCount, added shaderGroupHandleCaptureReplaySize)
  ◦ vkCmdTraceRaysNV → vkCmdTraceRaysKHR (params to struct)
  ◦ vkCreateRayTracingPipelinesNV → vkCreateRayTracingPipelinesKHR (different struct, changed functionality)
• Added enums, structures and commands:
  ◦ \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_NO\_NULL\_ANY\_HIT\_SHADERS\_BIT\_KHR}, \\
    \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_NO\_NULL\_CLOSEST\_HIT\_SHADERS\_BIT\_KHR}, \\
    \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_NO\_NULL\_MISS\_SHADERS\_BIT\_KHR}, \\
    \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_NO\_NULL\_INTERSECTION\_SHADERS\_BIT\_KHR}, \\
    \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_SKIP\__TRIANGLES\_BIT\_KHR}, \\
    \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_SKIP\_AABB\_S\_BIT\_KHR} to \texttt{VkPipelineCreateFlagBits}
  ◦ \texttt{VkPhysicalDeviceRayTracingPipelineFeaturesKHR} structure
  ◦ \texttt{VkDeviceOrHostAddressKHR} and \texttt{VkDeviceOrHostAddressConstKHR} unions
  ◦ \texttt{VkPipelineLibraryCreateInfoKHR} struct
  ◦ \texttt{VkRayTracingPipelineInterfaceCreateInfoKHR} struct
  ◦ \texttt{VkStridedDeviceAddressRegionKHR} struct
  ◦ \texttt{vkCmdTraceRaysIndirectKHR} command and \texttt{VkTraceRaysIndirectCommandKHR} struct
  ◦ \texttt{vkGetRayTracingCaptureReplayShaderGroupHandlesKHR} (shader group capture/replay)
  ◦ \texttt{vkCmdSetRayTracingPipelineStackSizeKHR} and \texttt{vkGetRayTracingShaderGroupStackSizeKHR} commands for stack size control

• Functionality removed:
  ◦ \texttt{VK\_PIPELINE\_CREATE\_DEFER\_COMPILE\_BIT\_NV}
  ◦ \texttt{vkCompileDeferredNV} command (replaced with \texttt{VK\_KHR\_deferred\_host\_operations})

(3) What are the changes between the public provisional (VK\_KHR\_ray\_tracing v8) release and the internal provisional (VK\_KHR\_ray\_tracing v9) release?

• Require Vulkan 1.1 and SPIR-V 1.4
• Added interactions with Vulkan 1.2 and \texttt{VK\_KHR\_vulkan\_memory\_model}
• added creation time capture and replay flags
  ◦ added \texttt{VK\_PIPELINE\_CREATE\_RAY\_TRACING\_SHADER\_GROUP\_HANDLE\_CAPTURE\_REPLAY\_BIT\_KHR} to \texttt{VkPipelineCreateFlagBits}
• replace \texttt{VkStridedBufferRegionKHR} with \texttt{VkStridedDeviceAddressRegionKHR} and change \texttt{vkCmdTraceRaysKHR}, \texttt{vkCmdTraceRaysIndirectKHR}, to take these for the shader binding table and use device addresses instead of buffers.
• require the shader binding table buffers to have the \texttt{VK\_BUFFER\_USAGE\_RAY\_TRACING\_BIT\_KHR} set
• make \texttt{VK\_KHR\_pipeline\_library} an interaction instead of required extension
• rename the \texttt{libraries} member of \texttt{VkRayTracingPipelineCreateInfoKHR} to \texttt{pLibraryInfo} and make it a pointer
• make \texttt{VK\_KHR\_deferred\_host\_operations} an interaction instead of a required extension (later went back on this)
• added explicit stack size management for ray tracing pipelines
  ◦ removed the \texttt{maxCallableSize} member of \texttt{VkRayTracingPipelineInterfaceCreateInfoKHR}
-added the `pDynamicState` member to `VkRayTracingPipelineCreateInfoKHR`
-
-added `VK_DYNAMIC_STATE_RAY_TRACING_PIPELINE_STACK_SIZE_KHR` dynamic state for ray tracing pipelines
-
-added `vkGetRayTracingShaderGroupStackSizeKHR` and `vkCmdSetRayTracingPipelineStackSizeKHR` commands
-
-added `VkShaderGroupShaderKHR` enum
-
-Added `maxRayDispatchInvocationCount` limit to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR`
-
-Added `shaderGroupHandleAlignment` property to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR`
-
-Added `maxRayHitAttributeSize` property to `VkPhysicalDeviceRayTracingPipelinePropertiesKHR`
-
-Clarify deferred host ops for pipeline creation
-
-`VkDeferredOperationKHR` is now a top-level parameter for `vkCreateRayTracingPipelinesKHR`
-
-removed `VkDeferredOperationInfoKHR` structure
-
-change deferred host creation/return parameter behavior such that the implementation can modify such parameters until the deferred host operation completes
-
-VK_KHR_deferred_host_operations is required again

(4) What are the changes between the internal provisional (VK_KHR_ray_tracing v9) release and the final (VK_KHR_acceleration_structure v11 / VK_KHR_ray_tracing_pipeline v1) release?

- refactor `VK_KHR_ray_tracing` into 3 extensions, enabling implementation flexibility and decoupling ray query support from ray pipelines:
  - `VK_KHR_acceleration_structure` (for acceleration structure operations)
  - `VK_KHR_ray_tracing_pipeline` (for ray tracing pipeline and shader stages)
  - `VK_KHR_ray_query` (for ray queries in existing shader stages)

- Require `Volatile` for the following builtins in the ray generation, closest hit, miss, intersection, and callable shader stages:
  - `SubgroupSize`, `SubgroupLocalInvocationId`, `SubgroupEqMask`, `SubgroupGeMask`, `SubgroupGtMask`, `SubgroupLeMask`, `SubgroupLtMask`
  - `SMIDNV`, `WarpIDNV`

- clarify buffer usage flags for ray tracing
  - `VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR` is added as an alias of `VK_BUFFER_USAGE_RAY_TRACING_BIT_NV` and is required on shader binding table buffers
  - `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT` is used in `VK_KHR_acceleration_structure` for `scratchData`

- rename `maxRecursionDepth` to `maxRayPipelineRecursionDepth` (pipeline creation) and `maxRayRecursionDepth` (limit) to reduce confusion

- Add queryable `maxRayHitAttributeSize` limit and rename members of
**VkRayTracingPipelineInterfaceCreateInfoKHR** to **maxPipelineRayPayloadSize** and **maxPipelineRayHitAttributeSize** for clarity

- Update SPIRV capabilities to use **RayTracingKHR**
- Extension is no longer provisional
- Define synchronization requirements for indirect trace rays and indirect buffer

(5) This extension adds gl_InstanceID for the intersection, any-hit, and closest hit shaders, but in KHR_vulkan_glsl, gl_InstanceID is replaced with gl_InstanceIndex. Which should be used for Vulkan in this extension?

**RESOLVED:** This extension uses gl_InstanceID and maps it to **InstanceId** in SPIR-V. It is acknowledged that this is different than other shader stages in Vulkan. There are two main reasons for the difference here:

- Symmetry with gl_PrimitiveID which is also available in these shaders
- There is no “baseInstance” relevant for these shaders, and so ID makes it more obvious that this is zero-based.

**Sample Code**

Example ray generation GLSL shader

```glsl
#version 450 core
#extension GL_EXT_ray_tracing : require
layout(set = 0, binding = 0, rgba8) uniform image2D image;
layout(set = 0, binding = 1) uniform accelerationStructureEXT as;
layout(location = 0) rayPayloadEXT float payload;

void main()
{
    vec4 col = vec4(0, 0, 0, 1);

    vec3 origin = vec3(float(gl_LaunchIDEXT.x)/float(gl_LaunchSizeEXT.x), float(gl_LaunchIDEXT.y)/float(gl_LaunchSizeEXT.y), 1.0);
    vec3 dir = vec3(0.0, 0.0, -1.0);

    traceRayEXT(as, 0, 0xff, 0, 1, 0, origin, 0.0, dir, 1000.0, 0);

    col.y = payload;

    imageStore(image, ivec2(gl_LaunchIDEXT.xy), col);
}
```

**Version History**

- Revision 1, 2020-11-12 (Mathieu Robart, Daniel Koch, Eric Werness, Tobias Hector)
  - Decomposition of the specification, from VK_KHR_ray_tracing to
VK_KHR_ray_tracing_pipeline (#1918,!3912)
◦ require certain subgroup and sm_shader_builtin shader builtins to be decorated as volatile in the ray generation, closest hit, miss, intersection, and callable stages (#1924,!3903,!3954)
◦ clarify buffer usage flags for ray tracing (#2181,!3939)
◦ rename maxRecursionDepth to maxRayPipelineRecursionDepth and maxRayRecursionDepth (#2203,!3937)
◦ add queriable maxRayHitAttributeSize and rename members of VkRayTracingPipelineInterfaceCreateInfoKHR (#2102,!3966)
◦ update to use RayTracingKHR SPIR-V capability
◦ add VUs for matching hit group type against geometry type (#2245,!3994)
◦ require RayTMaxKHR be volatile in intersection shaders (#2268,!4030)
◦ add numerical limits for ray parameters (#2235,!3960)
◦ fix SBT indexing rules for device addresses (#2308,!4079)
◦ relax formula for ray intersection candidate determination (#2322,!4080)
◦ add more details on ShaderRecordBufferKHR variables (#2230,!4083)
◦ clarify valid bits for InstanceCustomIndexKHR (GLSL/GLSL#19,!4128)
◦ allow at most one IncomingRayPayloadKHR, IncomingCallableDataKHR, and HitAttributeKHR (!4129)
◦ add minimum for maxShaderGroupStride (#2353,!4131)
◦ require VK_KHR_pipeline_library extension to be supported (#2348,!4135)
◦ clarify meaning of 'geometry index' (#2272,!4137)
◦ restrict traces to TLAS (#2239,!4141)
◦ add note about maxPipelineRayPayloadSize (#2383,!4172)
◦ do not require raygen shader in pipeline libraries (!4185)
◦ define sync for indirect trace rays and indirect buffer (#2407,!4208)

**VK_KHR_sampler_mirror_clamp_to_edge**

**Name String**

`VK_KHR_sampler_mirror_clamp_to_edge`

**Extension Type**

Device extension

**Registered Extension Number**

15

**Revision**

3
Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Tobias Hector @tobski

Other Extension Metadata

Last Modified Date

2019-08-17

Interactions and External Dependencies

- Promoted to Vulkan 1.2 Core

Contributors

- Tobias Hector, Imagination Technologies
- Jon Leech, Khronos

Description

VK_KHR_sampler_mirror_clamp_to_edge extends the set of sampler address modes to include an additional mode (VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE) that effectively uses a texture map twice as large as the original image in which the additional half of the new image is a mirror image of the original image.

This new mode relaxes the need to generate images whose opposite edges match by using the original image to generate a matching “mirror image”. This mode allows the texture to be mirrored only once in the negative s, t, and r directions.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2. However, if Vulkan 1.2 is supported and this extension is not, the VkSamplerAddressMode
VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE is optional. Since the original extension did not use an author suffix on the enum VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE, it is used by both core and extension implementations.

New Enum Constants

- VK_KHR_SAMPLER_MIRROR_CLAMP_TO_EDGE_EXTENSION_NAME
- VK_KHR_SAMPLER_MIRROR_CLAMP_TO_EDGE_SPEC_VERSION
- Extending VkSamplerAddressMode:
  - VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
Example

Creating a sampler with the new address mode in each dimension

```c
VkSamplerCreateInfo createInfo = 
{ 
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO // sType 
    // Other members set to application-desired values 
};

createInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE;
createInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE;
createInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE;

VkSampler sampler;
VkResult result = vkCreateSampler(
    device,
    &createInfo,
    &sampler);
```

Issues

1) Why are both KHR and core versions of the `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` token present?

**RESOLVED:** This functionality was intended to be required in Vulkan 1.0. We realized shortly before public release that not all implementations could support it, and moved the functionality into an optional extension, but did not apply the KHR extension suffix. Adding a KHR-suffixed alias of the non-suffixed enum has been done to comply with our own naming rules.

In a related change, before spec revision 1.1.121 this extension was hardwiring into the spec Makefile so it was always included with the Specification, even in the core-only versions. This has now been reverted, and it is treated as any other extension.

Version History

- Revision 1, 2016-02-16 (Tobias Hector)
  - Initial draft
- Revision 2, 2019-08-14 (Jon Leech)
  - Add KHR-suffixed alias of non-suffixed enum.
- Revision 3, 2019-08-17 (Jon Leech)
  - Add an issue explaining the reason for the extension API not being suffixed with KHR.

**VK_KHR_separate_depth_stencil_layouts**
Name String

VK_KHR_separate_depthStencil_layouts

Extension Type

Device extension

Registered Extension Number

242

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2
• Requires VK_KHR_create_renderpass2

Deprecation state

• Promoted to Vulkan 1.2

Contact

• Piers Daniell pdaniell-nv

Other Extension Metadata

Last Modified Date

2019-06-25

Interactions and External Dependencies

• Promoted to Vulkan 1.2 Core

Contributors

• Daniel Koch, NVIDIA
• Jeff Bolz, NVIDIA
• Jesse Barker, Unity
• Tobias Hector, AMD

Description

This extension allows image memory barriers for depth/stencil images to have just one of the 
VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT aspect bits set, rather than require both. This allows their layouts to be set independently. To support depth/stencil images with different layouts for the depth and stencil aspects, the depth/stencil attachment interface has been updated to support a separate layout for stencil.
Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending VkAttachmentDescription2:
  - VkAttachmentDescriptionStencilLayoutKHR
- Extending VkAttachmentReference2:
  - VkAttachmentReferenceStencilLayoutKHR
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceSeparateDepthStencilLayoutsFeaturesKHR

New Enum Constants

- VK_KHR_SEPARATE_DEPTH_STENCIL_LAYOUTS_EXTENSION_NAME
- VK_KHR_SEPARATE_DEPTH_STENCIL_LAYOUTS_SPEC_VERSION
- Extending VkImageLayout:
  - VK_IMAGE_LAYOUT_DEPTH_ATTACHMENT_OPTIMAL_KHR
  - VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR
  - VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR
  - VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_ATTACHMENT_DESCRIPTION_STENCIL_LAYOUT_KHR
  - VK_STRUCTURE_TYPE_ATTACHMENT_REFERENCE_STENCIL_LAYOUT_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SEPARATE_DEPTH_STENCIL_LAYOUTS_FEATURES_KHR

Version History

- Revision 1, 2019-06-25 (Piers Daniell)
  - Internal revisions

VK_KHR_shader_atomic_int64

Name String

VK_KHR_shader_atomic_int64

Extension Type

Device extension
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Aaron Hagan (@ahagan)

Other Extension Metadata

Last Modified Date

2018-07-05

Interactions and External Dependencies

- Promoted to Vulkan 1.2 Core
- This extension enables GL_ARB_gpu_shader_int64 and GL_EXT_shader_atomic_int64 for GLSL source languages.

Contributors

- Aaron Hagan, AMD
- Daniel Rakos, AMD
- Jeff Bolz, NVIDIA
- Neil Henning, Codeplay

Description

This extension advertises the SPIR-V Int64Atomics capability for Vulkan, which allows a shader to contain 64-bit atomic operations on signed and unsigned integers. The supported operations include OpAtomicMin, OpAtomicMax, OpAtomicAnd, OpAtomicOr, OpAtomicXor, OpAtomicAdd, OpAtomicExchange, and OpAtomicCompareExchange.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the shaderBufferInt64Atomics capability is optional. The original type, enum and command names are still available as aliases of the core functionality.
New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceShaderAtomicInt64FeaturesKHR

New Enum Constants

- VK_KHR_SHADER_ATOMIC_INT64_EXTENSION_NAME
- VK_KHR_SHADER_ATOMIC_INT64_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_INT64_FEATURES_KHR

New SPIR-V Capabilities

- Int64Atomics

Version History

- Revision 1, 2018-07-05 (Aaron Hagan)
  - Internal revisions

VK_KHR_shader_clock

Name String

VK_KHR_shader_clock

Extension Type

Device extension

Registered Extension Number

182

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Aaron Hagan

Other Extension Metadata

Last Modified Date

2019-4-25
IP Status
No known IP claims.

Interactions and External Dependencies
- This extension requires SPV_KHR_shader_clock.
- This extension provides API support for ARB_shader_clock and EXT_shader_realtime_clock

Contributors
- Aaron Hagan, AMD
- Daniel Koch, NVIDIA

Description
This extension advertises the SPIR-V ShaderClockKHR capability for Vulkan, which allows a shader to query a real-time or monotonically incrementing counter at the subgroup level or across the device level. The two valid SPIR-V scopes for OpReadClockKHR are Subgroup and Device.

When using GLSL source-based shading languages, the clockRealtime*EXT() timing functions map to the OpReadClockKHR instruction with a scope of Device, and the clock*ARB() timing functions map to the OpReadClockKHR instruction with a scope of Subgroup.

New Structures
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceShaderClockFeaturesKHR

New Enum Constants
- VK_KHR_SHADER_CLOCK_EXTENSION_NAME
- VK_KHR_SHADER_CLOCK_SPEC_VERSION
- Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CLOCK_FEATURES_KHR

New SPIR-V Capabilities
- ShaderClockKHR

Version History
- Revision 1, 2019-4-25 (Aaron Hagan)
  ◦ Initial revision

VK_KHR_shader_float16_int8

Name String
VK_KHR_shader_float16_int8
The **VK_KHR_shader_float16_int8** extension allows use of 16-bit floating-point types and 8-bit integer
types in shaders for arithmetic operations.

It introduces two new optional features `shaderFloat16` and `shaderInt8` which directly map to the `Float16` and the `Int8` SPIR-V capabilities. The `VK_KHR_shader_float16_int8` extension also specifies precision requirements for half-precision floating-point SPIR-V operations. This extension does not enable use of 8-bit integer types or 16-bit floating-point types in any shader input and output interfaces and therefore does not supersede the `VK_KHR_8bit_storage` or `VK_KHR_16bit_storage` extensions.

**Promotion to Vulkan 1.2**

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, both the `shaderFloat16` and `shaderInt8` capabilities are optional. The original type, enum and command names are still available as aliases of the core functionality.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceFloat16Int8FeaturesKHR`
  - `VkPhysicalDeviceShaderFloat16Int8FeaturesKHR`

**New Enum Constants**

- `VK_KHR_SHADER_FLOAT16_INT8_EXTENSION_NAME`
- `VK_KHR_SHADER_FLOAT16_INT8_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT16_INT8_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_FLOAT16_INT8_FEATURES_KHR`

**Version History**

- Revision 1, 2018-03-07 (Alexander Galazin)
  - Initial draft

**VK_KHR_shader_float_controls**

**Name String**

`VK_KHR_shader_float_controls`

**Extension Type**

Device extension

**Registered Extension Number**

198
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Alexander Galazin [alegal-arm](mailto:alegal-arm)

Other Extension Metadata

Last Modified Date

2018-09-11

Interactions and External Dependencies

- Promoted to Vulkan 1.2 Core
- This extension requires `SPV_KHR_float_controls`

IP Status

No known IP claims.

Contributors

- Alexander Galazin, Arm
- Jan-Harald Fredriksen, Arm
- Jeff Bolz, NVIDIA
- Graeme Leese, Broadcom
- Daniel Rakos, AMD

Description

The `VK_KHR_shader_float_controls` extension enables efficient use of floating-point computations through the ability to query and override the implementation’s default behavior for rounding modes, denormals, signed zero, and infinity.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending `VkPhysicalDeviceProperties2`:  

---

Appendix E: Layers & Extensions (Informative) | 2609
New Enums

- VkShaderFloatControlsIndependenceKHR

New Enum Constants

- VK_KHR_SHADER_FLOAT_CONTROLS_EXTENSION_NAME
- VK_KHR_SHADER_FLOAT_CONTROLS_SPEC_VERSION

Extending VkShaderFloatControlsIndependence:

- VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_32_BIT_ONLY_KHR
- VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_ALL_KHR
- VK_SHADER_FLOAT_CONTROLS_INDEPENDENCE_NONE_KHR

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FLOAT_CONTROLS_PROPERTIES_KHR

New SPIR-V Capabilities

- DenormPreserve
- DenormFlushToZero
- SignedZeroInfNanPreserve
- RoundingModeRTE
- RoundingModeRTZ

Issues

1) Which instructions must flush denorms?

**RESOLVED:** Only floating-point conversion, floating-point arithmetic, floating-point relational (except OpIsNaN, OpIsInf), and floating-point GLSL.std.450 extended instructions must flush denormals.

2) What is the denorm behavior for intermediate results?

**RESOLVED:** When a SPIR-V instruction is implemented as a sequence of other instructions:

- in the DenormFlushToZero execution mode, the intermediate instructions may flush denormals, the final result of the sequence must not be denormal.
- in the DenormPreserve execution mode, denormals must be preserved throughout the whole sequence.

3) Do denorm and rounding mode controls apply to OpSpecConstantOp?

**RESOLVED:** Yes, except when the opcode is OpQuantizeToF16.
4) The SPIR-V specification says that `OpConvertFToU` and `OpConvertFToS` unconditionally round towards zero. Do the rounding mode controls specified through the execution modes apply to them?

**RESOLVED**: No, these instructions unconditionally round towards zero.

5) Do any of the “Pack” GLSL.std.450 instructions count as conversion instructions and have the rounding mode applied?

**RESOLVED**: No, only instructions listed in “section 3.32.11. Conversion Instructions” of the SPIR-V specification count as conversion instructions.

6) When using inf/nan-ignore mode, what is expected of `OpIsNan` and `OpIsInf`?

**RESOLVED**: These instructions must always accurately detect inf/nan if it is passed to them.

### Version 4 API incompatibility

The original versions of `VK_KHR_shader_float_controls` shipped with booleans named “separateDenormSettings” and “separateRoundingModeSettings”, which at first glance could have indicated “they can all be set independently, or not”. However the spec language as written indicated that the 32-bit value could always be set independently, and only the 16- and 64-bit controls needed to be the same if these values were `VK_FALSE`.

As a result of this slight disparity, and lack of test coverage for this facet of the extension, we ended up with two different behaviors in the wild, where some implementations worked as written, and others worked based on the naming. As these are hard limits in hardware with reasons for exposure as written, it was not possible to standardise on a single way to make this work within the existing API.

No known users of this part of the extension exist in the wild, and as such the Vulkan WG took the unusual step of retroactively changing the once boolean value into a tri-state enum, breaking source compatibility. This was however done in such a way as to retain ABI compatibility, in case any code using this did exist; with the numerical values 0 and 1 retaining their original specified meaning, and a new value signifying the additional “all need to be set together” state. If any applications exist today, compiled binaries will continue to work as written in most cases, but will need changes before the code can be recompiled.

### Version History

- Revision 4, 2019-06-18 (Tobias Hector)
  - Modified settings restrictions, see **Version 4 API incompatibility**

- Revision 3, 2018-09-11 (Alexander Galazin)
  - Minor restructuring

- Revision 2, 2018-04-17 (Alexander Galazin)
  - Added issues and resolutions

- Revision 1, 2018-04-11 (Alexander Galazin)
VK_KHR_shader_non_semantic_info

Name String
VK_KHR_shader_non_semantic_info

Extension Type
Device extension

Registered Extension Number
294

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Baldur Karlsson 🌐baldurk

Other Extension Metadata

Last Modified Date
2019-10-16

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_KHR_non_semantic_info

Contributors
• Baldur Karlsson, Valve

Description
This extension allows the use of the SPV_KHR_non_semantic_info extension in SPIR-V shader modules.

New Enum Constants
• VK_KHR_SHADER_NON_SEMANTIC_INFO_EXTENSION_NAME
• VK_KHR_SHADER_NON_SEMANTIC_INFO_SPEC_VERSION

Version History
• Revision 1, 2019-10-16 (Baldur Karlsson)
VK_KHR_shader_subgroup_extended_types

Name String
VK_KHR_shader_subgroup_extended_types

Extension Type
Device extension

Registered Extension Number
176

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.1

Deprecation state
• Promoted to Vulkan 1.2

Contact
• Neil Henning sheredom

Other Extension Metadata

Last Modified Date
2019-01-08

IP Status
No known IP claims.

Interactions and External Dependencies
• Promoted to Vulkan 1.2 Core
• This extension provides API support for GLSL_EXT_shader_subgroup_extended_types

Contributors
• Jeff Bolz, NVIDIA
• Jan-Harald Fredriksen, Arm
• Neil Henning, AMD
• Daniel Koch, NVIDIA
• Jeff Leger, Qualcomm
• Graeme Leese, Broadcom
• David Neto, Google
Description

This extension enables the Non Uniform Group Operations in SPIR-V to support 8-bit integer, 16-bit integer, 64-bit integer, 16-bit floating-point, and vectors of these types.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceShaderSubgroupExtendedTypesFeaturesKHR

New Enum Constants

- VK_KHR_SHADER_SUBGROUP_EXTENDED_TYPES_EXTENSION_NAME
- VK_KHR_SHADER_SUBGROUP_EXTENDED_TYPES_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SUBGROUP_EXTENDED_TYPES_FEATURES_KHR

Version History

- Revision 1, 2019-01-08 (Neil Henning)
  - Initial draft

VK_KHR_shader_terminate_invocation

Name String

VK_KHR_shader_terminate_invocation

Extension Type

Device extension

Registered Extension Number

216

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2
Contact
• Jesse Hall

Last Modified Date
2020-08-11

IP Status
No known IP claims.

Interactions and External Dependencies
• Requires the SPV_KHR_terminate_invocation SPIR-V extension.

Contributors
• Alan Baker, Google
• Jeff Bolz, NVIDIA
• Jesse Hall, Google
• Ralph Potter, Samsung
• Tom Olson, Arm

Description
This extension adds Vulkan support for the SPV_KHR_terminate_invocation SPIR-V extension. That SPIR-V extension provides a new instruction, OpTerminateInvocation, which causes a shader invocation to immediately terminate and sets the coverage of shaded samples to 0; only previously executed instructions will have observable effects. The OpTerminateInvocation instruction, along with the OpDemoteToHelperInvocation instruction from the VK_EXT_shader_demote_to_helper_invocation extension, together replace the OpKill instruction, which could behave like either of these instructions. OpTerminateInvocation provides the behavior required by the GLSL discard statement, and should be used when available by GLSL compilers and applications that need the GLSL discard behavior.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceShaderTerminateInvocationFeaturesKHR

New Enum Constants
• VK_KHR_SHADER_TERMINATE_INVOCATION_EXTENSION_NAME
• VK_KHR_SHADER_TERMINATE_INVOCATION_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_TERMINATE_INVOCATION_FEATURES_KHR
Version History

• Revision 1, 2020-08-11 (Jesse Hall)

VK_KHR_shared_presentable_image

Name String

VK_KHR_shared_presentable_image

Extension Type

Device extension

Registered Extension Number

112

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_swapchain
• Requires VK_KHR_get_physical_device_properties2
• Requires VK_KHR_get_surface_capabilities2

Contact

• Alon Or-bach alonorbach

Other Extension Metadata

Last Modified Date

2017-03-20

IP Status

No known IP claims.

Contributors

• Alon Or-bach, Samsung Electronics
• Ian Elliott, Google
• Jesse Hall, Google
• Pablo Ceballos, Google
• Chris Forbes, Google
• Jeff Juliano, NVIDIA
• James Jones, NVIDIA
• Daniel Rakos, AMD
Description

This extension extends `VK_KHR_swapchain` to enable creation of a shared presentable image. This allows the application to use the image while the presentation engine is accessing it, in order to reduce the latency between rendering and presentation.

New Commands

• `vkGetSwapchainStatusKHR`

New Structures

• Extending `VkSurfaceCapabilities2KHR`:
  ◦ `VkSharedPresentSurfaceCapabilitiesKHR`

New Enum Constants

• `VK_KHR_SHARED_PRESENTABLE_IMAGE_EXTENSION_NAME`
• `VK_KHR_SHARED_PRESENTABLE_IMAGE_SPEC_VERSION`
• Extending `VkImageLayout`:
  ◦ `VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR`
• Extending `VkPresentModeKHR`:
  ◦ `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`
  ◦ `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`
• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_SHARED_PRESENT_SURFACE_CAPABILITIES_KHR`

Issues

1) Should we allow a Vulkan WSI swapchain to toggle between normal usage and shared presentation usage?

**Resolved**: No. WSI swapchains are typically recreated with new properties instead of having their properties changed. This can also save resources, assuming that fewer images are needed for shared presentation, and assuming that most VR applications do not need to switch between normal and shared usage.

2) Should we have a query for determining how the presentation engine refresh is triggered?
RESOLVED: Yes. This is done via which presentation modes a surface supports.

3) Should the object representing a shared presentable image be an extension of a VkSwapchainKHR or a separate object?

RESOLVED: Extension of a swapchain due to overlap in creation properties and to allow common functionality between shared and normal presentable images and swapchains.

4) What should we call the extension and the new structures it creates?

RESOLVED: Shared presentable image / shared present.

5) Should the minImageCount and presentMode values of the VkSwapchainCreateInfoKHR be ignored, or required to be compatible values?

RESOLVED: minImageCount must be set to 1, and presentMode should be set to either VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR or VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR.

6) What should the layout of the shared presentable image be?

RESOLVED: After acquiring the shared presentable image, the application must transition it to the VK_IMAGE_LAYOUT_SHARED_PRESENT_KHR layout prior to it being used. After this initial transition, any image usage that was requested during swapchain creation can be performed on the image without layout transitions being performed.

7) Do we need a new API for the trigger to refresh new content?

RESOLVED: vkQueuePresentKHR to act as API to trigger a refresh, as will allow combination with other compatible extensions to vkQueuePresentKHR.

8) How should an application detect a VK_ERROR_OUT_OF_DATE_KHR error on a swapchain using the VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR present mode?

RESOLVED: Introduce vkGetSwapchainStatusKHR to allow applications to query the status of a swapchain using a shared presentation mode.

9) What should subsequent calls to vkQueuePresentKHR for VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR swapchains be defined to do?

RESOLVED: State that implementations may use it as a hint for updated content.

10) Can the ownership of a shared presentable image be transferred to a different queue?

RESOLVED: No. It is not possible to transfer ownership of a shared presentable image obtained from a swapchain created using VK_SHARING_MODE_EXCLUSIVE after it has been presented.

11) How should vkQueueSubmit behave if a command buffer uses an image from a VK_ERROR_OUT_OF_DATE_KHR swapchain?

RESOLVED: vkQueueSubmit is expected to return the VK_ERROR_DEVICE_LOST error.

12) Can Vulkan provide any guarantee on the order of rendering, to enable beam chasing?
RESOLVED: This could be achieved via use of render passes to ensure strip rendering.

Version History

• Revision 1, 2017-03-20 (Alon Or-bach)
  ◦ Internal revisions

VK_KHR_spirv_1_4

Name String

VK_KHR_spirv_1_4

Extension Type

Device extension

Registered Extension Number

237

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.1
  • Requires VK_KHR_shader_float_controls

Deprecation state

• Promoted to Vulkan 1.2

Contact

• Jesse Hall @critsec

Other Extension Metadata

Last Modified Date

2019-04-01

IP Status

No known IP claims.

Interactions and External Dependencies

• Requires SPIR-V 1.4.
  • Promoted to Vulkan 1.2 Core

Contributors

• Alexander Galazin, Arm
  • David Neto, Google
Description

This extension allows the use of SPIR-V 1.4 shader modules. SPIR-V 1.4’s new features primarily make it an easier target for compilers from high-level languages, rather than exposing new hardware functionality.

SPIR-V 1.4 incorporates features that are also available separately as extensions. SPIR-V 1.4 shader modules do not need to enable those extensions with the `OpExtension` opcode, since they are integral parts of SPIR-V 1.4.

SPIR-V 1.4 introduces new floating point execution mode capabilities, also available via `SPV_KHR_float_controls`. Implementations are not required to support all of these new capabilities; support can be queried using `VkPhysicalDeviceFloatControlsPropertiesKHR` from the `VK_KHR_shader_float_controls` extension.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Enum Constants

- `VK_KHR_SPIRV_1_4_EXTENSION_NAME`
- `VK_KHR_SPIRV_1_4_SPEC_VERSION`

Issues

1. Should we have an extension specific to this SPIR-V version, or add a version-generic query for SPIR-V version? SPIR-V 1.4 doesn't need any other API changes.

**RESOLVED:** Just expose SPIR-V 1.4.

Most new SPIR-V versions introduce optionally-required capabilities or have implementation-defined limits, and would need more API and specification changes specific to that version to make them available in Vulkan. For example, to support the subgroup capabilities added in SPIR-V 1.3 required introducing `VkPhysicalDeviceSubgroupProperties` to allow querying the supported group operation categories, maximum supported subgroup size, etc. While we could expose the parts of a new SPIR-V version that don't need accompanying changes generically, we'll still end up writing extensions specific to each version for the remaining parts. Thus the generic mechanism won't reduce future spec-writing effort. In addition, making it clear which parts of a future version are supported by the generic mechanism and which can't be used without specific support would be difficult to get right ahead of time.
2. Can different stages of the same pipeline use shaders with different SPIR-V versions?

**RESOLVED:** Yes.

Mixing SPIR-V versions 1.0-1.3 in the same pipeline has not been disallowed, so it would be inconsistent to disallow mixing 1.4 with previous versions. SPIR-V 1.4 does not introduce anything that should cause new difficulties here.

3. Must Vulkan extensions corresponding to SPIR-V extensions that were promoted to core in 1.4 be enabled in order to use that functionality in a SPIR-V 1.4 module?

**RESOLVED:** No, with caveats.

The SPIR-V 1.4 module does not need to declare the SPIR-V extensions, since the functionality is now part of core, so there is no need to enable the Vulkan extension that allows SPIR-V modules to declare the SPIR-V extension. However, when the functionality that is now core in SPIR-V 1.4 is optionally supported, the query for support is provided by a Vulkan extension, and that query can only be used if the extension is enabled.

This applies to any SPIR-V version; specifically for SPIR-V 1.4 this only applies to the functionality from `SPV_KHR_float_controls`, which was made available in Vulkan by `VK_KHR_shader_float_controls`. Even though the extension was promoted in SPIR-V 1.4, the capabilities are still optional in implementations that support `VK_KHR_spirv_1_4`.

A SPIR-V 1.4 module doesn't need to enable `SPV_KHR_float_controls` in order to use the capabilities, so if the application has a priori knowledge that the implementation supports the capabilities, it doesn't need to enable `VK_KHR_shader_float_controls`. However, if it doesn't have this knowledge and has to query for support at runtime, it must enable `VK_KHR_shader_float_controls` in order to use `VkPhysicalDeviceFloatControlsPropertiesKHR`.

**Version History**

- Revision 1, 2019-04-01 (Jesse Hall)
  - Internal draft versions

**VK_KHR_surface**

**Name String**

`VK_KHR_surface`

**Extension Type**

Instance extension

**Registered Extension Number**

1

**Revision**

25
Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- James Jones  
cubanismo
- Ian Elliott  
ianelliottus

Other Extension Metadata

Last Modified Date
2016-08-25

IP Status
No known IP claims.

Contributors

- Patrick Doane, Blizzard
- Ian Elliott, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Jeff Vigil, Qualcomm
- Chia-I Wu, LunarG
- Jason Ekstrand, Intel

Description

The **VK_KHR_surface** extension is an instance extension. It introduces *VkSurfaceKHR* objects, which abstract native platform surface or window objects for use with Vulkan. It also provides a way to determine whether a queue family in a physical device supports presenting to particular surface.

Separate extensions for each platform provide the mechanisms for creating *VkSurfaceKHR* objects, but once created they may be used in this and other platform-independent extensions, in particular the **VK_KHR_swapchain** extension.

New Object Types

- *VkSurfaceKHR*
New Commands

- vkDestroySurfaceKHR
- vkGetPhysicalDeviceSurfaceCapabilitiesKHR
- vkGetPhysicalDeviceSurfaceFormatsKHR
- vkGetPhysicalDeviceSurfacePresentModesKHR
- vkGetPhysicalDeviceSurfaceSupportKHR

New Structures

- VkSurfaceCapabilitiesKHR
- VkSurfaceFormatKHR

New Enums

- VkColorSpaceKHR
- VkCompositeAlphaFlagBitsKHR
- VkPresentModeKHR
- VkSurfaceTransformFlagBitsKHR

New Bitmasks

- VkCompositeAlphaFlagsKHR

New Enum Constants

- VK_KHR_SURFACE_EXTENSION_NAME
- VK_KHR_SURFACE_SPEC_VERSION

Extending VkObjectType:

- VK_OBJECT_TYPE_SURFACE_KHR

Extending VkResult:

- VK_ERROR_NATIVE_WINDOW_IN_USE_KHR
- VK_ERROR_SURFACE_LOST_KHR

Examples

Note

The example code for the VK_KHR_surface and VK_KHR_swapchain extensions was removed from the appendix after revision 1.0.29. This WSI example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c).
Issues

1) Should this extension include a method to query whether a physical device supports presenting to a specific window or native surface on a given platform?

RESOLVED: Yes. Without this, applications would need to create a device instance to determine whether a particular window can be presented to. Knowing that a device supports presentation to a platform in general is not sufficient, as a single machine might support multiple seats, or instances of the platform that each use different underlying physical devices. Additionally, on some platforms, such as the X Window System, different drivers and devices might be used for different windows depending on which section of the desktop they exist on.

2) Should the vkGetPhysicalDeviceSurfaceCapabilitiesKHR, vkGetPhysicalDeviceSurfaceFormatsKHR, and vkGetPhysicalDeviceSurfacePresentModesKHR functions be in this extension and operate on physical devices, rather than being in Vk_KHR_swapchain (i.e. device extension) and being dependent on VkDevice?

RESOLVED: Yes. While it might be useful to depend on VkDevice (and therefore on enabled extensions and features) for the queries, Vulkan was released only with the VkPhysicalDevice versions. Many cases can be resolved by a Valid Usage statement, and/or by a separate pNext chain version of the query struct specific to a given extension or parameters, via extensible versions of the queries: vkGetPhysicalDeviceSurfaceCapabilities2KHR, vkGetPhysicalDeviceSurfaceFormats2KHR, and vkGetPhysicalDeviceSurfacePresentModes2EXT.

3) Should Vulkan support Xlib or XCB as the API for accessing the X Window System platform?

RESOLVED: Both. XCB is a more modern and efficient API, but Xlib usage is deeply ingrained in many applications and likely will remain in use for the foreseeable future. Not all drivers necessarily need to support both, but including both as options in the core specification will probably encourage support, which should in turn ease adoption of the Vulkan API in older codebases. Additionally, the performance improvements possible with XCB likely will not have a measurable impact on the performance of Vulkan presentation and other minimal window system interactions defined here.

4) Should the GBM platform be included in the list of platform enums?

RESOLVED: Deferred, and will be addressed with a platform-specific extension to be written in the future.

Version History

- Revision 1, 2015-05-20 (James Jones)
  - Initial draft, based on LunarG KHR spec, other KHR specs, patches attached to bugs.
- Revision 2, 2015-05-22 (Ian Elliott)
  - Created initial Description section.
  - Removed query for whether a platform requires the use of a queue for presentation, since it was decided that presentation will always be modeled as being part of the queue.
  - Fixed typos and other minor mistakes.
• Revision 3, 2015-05-26 (Ian Elliott)
  ◦ Improved the Description section.

• Revision 4, 2015-05-27 (James Jones)
  ◦ Fixed compilation errors in example code.

• Revision 5, 2015-06-01 (James Jones)
  ◦ Added issues 1 and 2 and made related spec updates.

• Revision 6, 2015-06-01 (James Jones)
  ◦ Merged the platform type mappings table previously removed from VK_KHR_swapchain with the platform description table in this spec.
  ◦ Added issues 3 and 4 documenting choices made when building the initial list of native platforms supported.

• Revision 7, 2015-06-11 (Ian Elliott)
  ◦ Updated table 1 per input from the KHR TSG.
  ◦ Updated issue 4 (GBM) per discussion with Daniel Stone. He will create a platform-specific extension sometime in the future.

• Revision 8, 2015-06-17 (James Jones)
  ◦ Updated enum-extending values using new convention.
  ◦ Fixed the value of VK_SURFACE_PLATFORM_INFO_TYPE_SUPPORTED_KHR.

• Revision 9, 2015-06-17 (James Jones)
  ◦ Rebased on Vulkan API version 126.

• Revision 10, 2015-06-18 (James Jones)
  ◦ Marked issues 2 and 3 resolved.

• Revision 11, 2015-06-23 (Ian Elliott)
  ◦ Examples now show use of function pointers for extension functions.
  ◦ Eliminated extraneous whitespace.

• Revision 12, 2015-07-07 (Daniel Rakos)
  ◦ Added error section describing when each error is expected to be reported.
  ◦ Replaced the term “queue node index” with “queue family index” in the spec as that is the agreed term to be used in the latest version of the core header and spec.
  ◦ Replaced bool32_t with VkBool32.

• Revision 13, 2015-08-06 (Daniel Rakos)
  ◦ Updated spec against latest core API header version.

• Revision 14, 2015-08-20 (Ian Elliott)
  ◦ Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
  ◦ Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the
header file.
  ◦ Did miscellaneous cleanup, etc.

• Revision 15, 2015-08-20 (Ian Elliott—porting a 2015-07-29 change from James Jones)
  ◦ Moved the surface transform enums here from VK_WSI_swapchain so they could be reused by VK_WSI_display.

• Revision 16, 2015-09-01 (James Jones)
  ◦ Restore single-field revision number.

• Revision 17, 2015-09-01 (James Jones)
  ◦ Fix example code compilation errors.

• Revision 18, 2015-09-26 (Jesse Hall)
  ◦ Replaced VkSurfaceDescriptionKHR with the VkSurfaceKHR object, which is created via layered extensions. Added VkDestroySurfaceKHR.

• Revision 19, 2015-09-28 (Jesse Hall)
  ◦ Renamed from VK_EXT_KHR_swapchain to VK_EXT_KHR_surface.

• Revision 20, 2015-09-30 (Jeff Vigil)
  ◦ Add error result VK_ERROR_SURFACE_LOST_KHR.

• Revision 21, 2015-10-15 (Daniel Rakos)
  ◦ Updated the resolution of issue #2 and include the surface capability queries in this extension.
  ◦ Renamed SurfaceProperties to SurfaceCapabilities as it better reflects that the values returned are the capabilities of the surface on a particular device.
  ◦ Other minor cleanup and consistency changes.

• Revision 22, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_surface to VK_KHR_surface.

• Revision 23, 2015-11-03 (Daniel Rakos)
  ◦ Added allocation callbacks to vkDestroySurfaceKHR.

• Revision 24, 2015-11-10 (Jesse Hall)
  ◦ Removed VkSurfaceTransformKHR. Use VkSurfaceTransformFlagBitsKHR instead.
  ◦ Rename VkSurfaceCapabilitiesKHR member maxImageArraySize to maxImageArrayLayers.

• Revision 25, 2016-01-14 (James Jones)
  ◦ Moved VK_ERROR_NATIVE_WINDOW_IN_USE_KHR from the VK_KHR_android_surface to the VK_KHR_surface extension.

• 2016-08-23 (Ian Elliott)
  ◦ Update the example code, to not have so many characters per line, and to split out a new example to show how to obtain function pointers.

• 2016-08-25 (Ian Elliott)
VK_KHR_surface_protected_capabilities

Name String
VK_KHR_surface_protected_capabilities

Extension Type
Instance extension

Registered Extension Number
240

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.1
• Requires VK_KHR_get_surface_capabilities2

Contact
• Sandeep Shinde @sashinde

Other Extension Metadata

Last Modified Date
2018-12-18

IP Status
No known IP claims.

Contributors
• Sandeep Shinde, NVIDIA
• James Jones, NVIDIA
• Daniel Koch, NVIDIA

Description
This extension extends VkSurfaceCapabilities2KHR, providing applications a way to query whether swapchains can be created with the VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR flag set.

Vulkan 1.1 added (optional) support for protect memory and protected resources including buffers (VK_BUFFER_CREATE_PROTECTED_BIT), images (VK_IMAGE_CREATE_PROTECTED_BIT), and swapchains (VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR). However, on implementations which support multiple windowing systems, not all window systems may be able to provide a protected display path.
This extension provides a way to query if a protected swapchain created for a surface (and thus a specific windowing system) can be displayed on screen. It extends the existing VkSurfaceCapabilities2KHR structure with a new VkSurfaceProtectedCapabilitiesKHR structure from which the application can obtain information about support for protected swapchain creation through vkGetPhysicalDeviceSurfaceCapabilities2KHR.

New Structures

- Extending VkSurfaceCapabilities2KHR:
  - VkSurfaceProtectedCapabilitiesKHR

New Enum Constants

- VK_KHR_SURFACE_PROTECTED_CAPABILITIES_EXTENSION_NAME
- VK_KHR_SURFACE_PROTECTED_CAPABILITIES_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_SURFACE_PROTECTED_CAPABILITIES_KHR

Version History

- Revision 1, 2018-12-18 (Sandeep Shinde, Daniel Koch)
  - Internal revisions.

**VK_KHR_swapchain**

**Name String**

VK_KHR_swapchain

**Extension Type**

Device extension

**Registered Extension Number**

2

**Revision**

70

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_surface

**Contact**

- James Jones cubanismo
- Ian Elliott ianelliottus
Description

The VK_KHR_swapchain extension is the device-level companion to the VK_KHR_surface extension. It introduces VkSwapchainKHR objects, which provide the ability to present rendering results to a surface.

New Object Types

• VkSwapchainKHR

New Commands

• vkAcquireNextImageKHR
• vkCreateSwapchainKHR
• vkDestroySwapchainKHR
• vkGetSwapchainImagesKHR
• vkQueuePresentKHR

If Version 1.1 is supported:

• vkAcquireNextImage2KHR
• vkGetDeviceGroupPresentCapabilitiesKHR
• vkGetDeviceGroupSurfacePresentModesKHR
• vkGetPhysicalDevicePresentRectanglesKHR

New Structures

• VkPresentInfoKHR
• VkSwapchainCreateInfoKHR

If Version 1.1 is supported:

• VkAcquireNextImageInfoKHR
• VkDeviceGroupPresentCapabilitiesKHR
• Extending VkBindImageMemoryInfo:
  ◦ VkBindImageMemorySwapchainInfoKHR
• Extending VkImageCreateInfo:
  ◦ VkImageSwapchainCreateInfoKHR
• Extending VkPresentInfoKHR:
  ◦ VkDeviceGroupPresentInfoKHR
• Extending VkSwapchainCreateInfoKHR:
  ◦ VkDeviceGroupSwapchainCreateInfoKHR

New Enums

• VkSwapchainCreateFlagBitsKHR

If Version 1.1 is supported:

• VkDeviceGroupPresentModeFlagBitsKHR

New Bitmasks

• VkSwapchainCreateFlagsKHR

If Version 1.1 is supported:

• VkDeviceGroupPresentModeFlagsKHR
New Enum Constants

- `VK_KHR_SWAPCHAIN_EXTENSION_NAME`
- `VK_KHR_SWAPCHAIN_SPEC_VERSION`

Extending `VkImageLayout`:
- `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`

Extending `VkObjectType`:
- `VK_OBJECT_TYPE_SWAPCHAIN_KHR`

Extending `VkResult`:
- `VK_ERROR_OUT_OF_DATE_KHR`
- `VK_SUBOPTIMAL_KHR`

Extending `VkStructureType`:
- `VK_STRUCTURE_TYPE_PRESENT_INFO_KHR`
- `VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR`

If Version 1.1 is supported:

Extending `VkStructureType`:
- `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`
- `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`
- `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR`
- `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR`
- `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
- `VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR`

Extending `VkSwapchainCreateFlagBitsKHR`:
- `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR`
- `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`

Issues

1) Does this extension allow the application to specify the memory backing of the presentable images?

**RESOLVED:** No. Unlike standard images, the implementation will allocate the memory backing of the presentable image.

2) What operations are allowed on presentable images?

**RESOLVED:** This is determined by the image usage flags specified when creating the presentable image’s swapchain.

3) Does this extension support MSAA presentable images?
**RESOLVED:** No. Presentable images are always single-sampled. Multi-sampled rendering must use regular images. To present the rendering results the application must manually resolve the multi-sampled image to a single-sampled presentable image prior to presentation.

4) Does this extension support stereo/multi-view presentable images?

**RESOLVED:** Yes. The number of views associated with a presentable image is determined by the `imageArrayLayers` specified when creating a swapchain. All presentable images in a given swapchain use the same array size.

5) Are the layers of stereo presentable images half-sized?

**RESOLVED:** No. The image extents always match those requested by the application.

6) Do the “present” and “acquire next image” commands operate on a queue? If not, do they need to include explicit semaphore objects to interlock them with queue operations?

**RESOLVED:** The present command operates on a queue. The image ownership operation it represents happens in order with other operations on the queue, so no explicit semaphore object is required to synchronize its actions.

Applications may want to acquire the next image in separate threads from those in which they manage their queue, or in multiple threads. To make such usage easier, the acquire next image command takes a semaphore to signal as a method of explicit synchronization. The application must later queue a wait for this semaphore before queuing execution of any commands using the image.

7) Does `vkAcquireNextImageKHR` block if no images are available?

**RESOLVED:** The command takes a timeout parameter. Special values for the timeout are 0, which makes the call a non-blocking operation, and `UINT64_MAX`, which blocks indefinitely. Values in between will block for up to the specified time. The call will return when an image becomes available or an error occurs. It may, but is not required to, return before the specified timeout expires if the swapchain becomes out of date.

8) Can multiple presents be queued using one `vkQueuePresentKHR` call?

**RESOLVED:** Yes. `VkPresentInfoKHR` contains a list of swapchains and corresponding image indices that will be presented. When supported, all presentations queued with a single `vkQueuePresentKHR` call will be applied atomically as one operation. The same swapchain must not appear in the list more than once. Later extensions may provide applications stronger guarantees of atomicity for such present operations, and/or allow them to query whether atomic presentation of a particular group of swapchains is possible.

9) How do the presentation and acquire next image functions notify the application the targeted surface has changed?

**RESOLVED:** Two new result codes are introduced for this purpose:

- **VK_SUBOPTIMAL_KHR** - Presentation will still succeed, subject to the window resize behavior, but the swapchain is no longer configured optimally for the surface it targets. Applications should
query updated surface information and recreate their swapchain at the next convenient opportunity.

- **VK_ERROR_OUT_OF_DATE_KHR** - Failure. The swapchain is no longer compatible with the surface it targets. The application must query updated surface information and recreate the swapchain before presentation will succeed.

These can be returned by both `vkAcquireNextImageKHR` and `vkQueuePresentKHR`.

10) Does the `vkAcquireNextImageKHR` command return a semaphore to the application via an output parameter, or accept a semaphore to signal from the application as an object handle parameter?

**RESOLVED:** Accept a semaphore to signal as an object handle. This avoids the need to specify whether the application must destroy the semaphore or whether it is owned by the swapchain, and if the latter, what its lifetime is and whether it can be reused for other operations once it is received from `vkAcquireNextImageKHR`.

11) What types of swapchain queuing behavior should be exposed? Options include swap interval specification, mailbox/most recent vs. FIFO queue management, targeting specific vertical blank intervals or absolute times for a given present operation, and probably others. For some of these, whether they are specified at swapchain creation time or as per-present parameters needs to be decided as well.

**RESOLVED:** The base swapchain extension will expose 3 possible behaviors (of which, FIFO will always be supported):

- **Immediate present:** Does not wait for vertical blanking period to update the current image, likely resulting in visible tearing. No internal queue is used. Present requests are applied immediately.

- **Mailbox queue:** Waits for the next vertical blanking period to update the current image. No tearing should be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for reuse by the application.

- **FIFO queue:** Waits for the next vertical blanking period to update the current image. No tearing should be observed. An internal queue containing `numSwapchainImages - 1` entries is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty.

Not all surfaces will support all of these modes, so the modes supported will be returned using a surface info query. All surfaces must support the FIFO queue mode. Applications must choose one of these modes up front when creating a swapchain. Switching modes can be accomplished by recreating the swapchain.

12) Can `VK_PRESENT_MODE_MAILBOX_KHR` provide non-blocking guarantees for `vkAcquireNextImageKHR`? If so, what is the proper criteria?

**RESOLVED:** Yes. The difficulty is not immediately obvious here. Naively, if at least 3 images are
requested, mailbox mode should always have an image available for the application if the application does not own any images when the call to `vkAcquireNextImageKHR` was made. However, some presentation engines may have more than one “current” image, and would still need to block in some cases. The right requirement appears to be that if the application allocates the surface’s minimum number of images + 1 then it is guaranteed non-blocking behavior when it does not currently own any images.

13) Is there a way to create and initialize a new swapchain for a surface that has generated a `VK_SUBOPTIMAL_KHR` return code while still using the old swapchain?

**RESOLVED:** Not as part of this specification. This could be useful to allow the application to create an “optimal” replacement swapchain and rebuild all its command buffers using it in a background thread at a low priority while continuing to use the “suboptimal” swapchain in the main thread. It could probably use the same “atomic replace” semantics proposed for recreating direct-to-device swapchains without incurring a mode switch. However, after discussion, it was determined some platforms probably could not support concurrent swapchains for the same surface though, so this will be left out of the base KHR extensions. A future extension could add this for platforms where it is supported.

14) Should there be a special value for `VkSurfaceCapabilitiesKHR::maxImageCount` to indicate there are no practical limits on the number of images in a swapchain?

**RESOLVED:** Yes. There will often be cases where there is no practical limit to the number of images in a swapchain other than the amount of available resources (i.e., memory) in the system. Trying to derive a hard limit from things like memory size is prone to failure. It is better in such cases to leave it to applications to figure such soft limits out via trial/failure iterations.

15) Should there be a special value for `VkSurfaceCapabilitiesKHR::maxImageExtent` to indicate there is no practical limit on the surface size?

**RESOLVED:** No. It seems unlikely such a system would exist. 0 could be used to indicate the platform places no limits on the extents beyond those imposed by Vulkan for normal images, but this query could just as easily return those same limits, so a special “unlimited” value does not seem useful for this field.

17) How should surface rotation and mirroring be exposed to applications? How do they specify rotation and mirroring transforms applied prior to presentation?

**RESOLVED:** Applications can query both the supported and current transforms of a surface. Both are specified relative to the device’s “natural” display rotation and direction. The supported transforms indicate which orientations the presentation engine accepts images in. For example, a presentation engine that does not support transforming surfaces as part of presentation, and which is presenting to a surface that is displayed with a 90-degree rotation, would return only one supported transform bit: `VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR`. Applications must transform
their rendering by the transform they specify when creating the swapchain in preTransform field.

18) Can surfaces ever not support VK_MIRROR_NONE? Can they support vertical and horizontal mirroring simultaneously? Relatedly, should VK_MIRROR_NONE[_BIT] be zero, or bit one, and should applications be allowed to specify multiple pre and current mirror transform bits, or exactly one?

**RESOLVED:** Since some platforms may not support presenting with a transform other than the native window’s current transform, and prerotation/mirroring are specified relative to the device’s natural rotation and direction, rather than relative to the surface’s current rotation and direction, it is necessary to express lack of support for no mirroring. To allow this, the MIRROR_NONE enum must occupy a bit in the flags. Since MIRROR_NONE must be a bit in the bitmask rather than a bitmask with no values set, allowing more than one bit to be set in the bitmask would make it possible to describe undefined transforms such as VK_MIRROR_NONE_BIT | VK_MIRROR_HORIZONTAL_BIT, or a transform that includes both “no mirroring” and “horizontal mirroring” simultaneously. Therefore, it is desirable to allow specifying all supported mirroring transforms using only one bit. The question then becomes, should there be a VK_MIRROR_HORIZONTAL_AND_VERTICAL_BIT to represent a simultaneous horizontal and vertical mirror transform? However, such a transform is equivalent to a 180 degree rotation, so presentation engines and applications that wish to support or use such a transform can express it through rotation instead. Therefore, 3 exclusive bits are sufficient to express all needed mirroring transforms.

19) Should support for sRGB be required?

**RESOLVED:** In the advent of UHD and HDR display devices, proper color space information is vital to the display pipeline represented by the swapchain. The app can discover the supported format/color-space pairs and select a pair most suited to its rendering needs. Currently only the sRGB color space is supported, future extensions may provide support for more color spaces. See issues 23 and 24.

20) Is there a mechanism to modify or replace an existing swapchain with one targeting the same surface?

**RESOLVED:** Yes. This is described above in the text.

21) Should there be a way to set prerotation and mirroring using native APIs when presenting using a Vulkan swapchain?

**RESOLVED:** Yes. The transforms that can be expressed in this extension are a subset of those possible on native platforms. If a platform exposes a method to specify the transform of presented images for a given surface using native methods and exposes more transforms or other properties for surfaces than Vulkan supports, it might be impossible, difficult, or inconvenient to set some of those properties using Vulkan KHR extensions and some using the native interfaces. To avoid overwriting properties set using native commands when presenting using a Vulkan swapchain, the application can set the pretransform to “inherit”, in which case the current native properties will be used, or if none are available, a platform-specific default will be used. Platforms that do not specify a reasonable default or do not provide native mechanisms to specify such transforms should not include the inherit bits in the supportedTransforms bitmask they return in VkSurfaceCapabilitiesKHR.

22) Should the content of presentable images be clipped by objects obscuring their target surface?
Applications can choose which behavior they prefer. Allowing the content to be clipped could enable more efficient presentation methods on some platforms, but some applications might rely on the content of presentable images to perform techniques such as partial updates or motion blur.

23) What is the purpose of specifying a VkColorSpaceKHR along with VkFormat when creating a swapchain?

**RESOLVED:** While Vulkan itself is color space agnostic (e.g. even the meaning of R, G, B and A can be freely defined by the rendering application), the swapchain eventually will have to present the images on a display device with specific color reproduction characteristics. If any color space transformations are necessary before an image can be displayed, the color space of the presented image must be known to the swapchain. A swapchain will only support a restricted set of color format and -space pairs. This set can be discovered via vkGetPhysicalDeviceSurface FormatsKHR. As it can be expected that most display devices support the sRGB color space, at least one format/color-space pair has to be exposed, where the color space is VK_COLOR_SPACE_SRGB_NONLINEAR_KHR.

24) How are sRGB formats and the sRGB color space related?

**RESOLVED:** While Vulkan exposes a number of SRGB texture formats, using such formats does not guarantee working in a specific color space. It merely means that the hardware can directly support applying the non-linear transfer functions defined by the sRGB standard color space when reading from or writing to images of those formats. Still, it is unlikely that a swapchain will expose a *_SRGB format along with any color space other than VK_COLOR_SPACE_SRGB_NONLINEAR_KHR.

On the other hand, non-_SRGB formats will be very likely exposed in pair with a SRGB color space. This means, the hardware will not apply any transfer function when reading from or writing to such images, yet they will still be presented on a device with sRGB display characteristics. In this case the application is responsible for applying the transfer function, for instance by using shader math.

25) How are the lifetimes of surfaces and swapchains targeting them related?

**RESOLVED:** A surface must outlive any swapchains targeting it. A VkSurfaceKHR owns the binding of the native window to the Vulkan driver.

26) How can the client control the way the alpha channel of swapchain images is treated by the presentation engine during compositing?

**RESOLVED:** We should add new enum values to allow the client to negotiate with the presentation engine on how to treat image alpha values during the compositing process. Since not all platforms can practically control this through the Vulkan driver, a value of VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR is provided like for surface transforms.

27) Is vkCreateSwapchainKHR the right function to return VK_ERROR_NATIVE_WINDOW_IN_USE_KHR, or should the various platform-specific VkSurfaceKHR factory functions catch this error earlier?

**RESOLVED:** For most platforms, the VkSurfaceKHR structure is a simple container holding the data that identifies a native window or other object representing a surface on a particular platform. For the surface factory functions to return this error, they would likely need to register a reference on
the native objects with the native display server somehow, and ensure no other such references exist. Surfaces were not intended to be that heavyweight.

Swapchains are intended to be the objects that directly manipulate native windows and communicate with the native presentation mechanisms. Swapchains will already need to communicate with the native display server to negotiate allocation and/or presentation of presentable images for a native surface. Therefore, it makes more sense for swapchain creation to be the point at which native object exclusivity is enforced. Platforms may choose to enforce further restrictions on the number of \texttt{VkSurfaceKHR} objects that may be created for the same native window if such a requirement makes sense on a particular platform, but a global requirement is only sensible at the swapchain level.

**Examples**

\begin{quote}
\textit{Note}

The example code for the \texttt{VK_KHR_surface} and \texttt{VK_KHR_swapchain} extensions was removed from the appendix after revision 1.0.29. This WSI example code was ported to the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c).
\end{quote}

**Version History**

- Revision 1, 2015-05-20 (James Jones)
  - Initial draft, based on LunarG KHR spec, other KHR specs, patches attached to bugs.
- Revision 2, 2015-05-22 (Ian Elliott)
  - Made many agreed-upon changes from 2015-05-21 KHR TSG meeting. This includes using only a queue for presentation, and having an explicit function to acquire the next image.
  - Fixed typos and other minor mistakes.
- Revision 3, 2015-05-26 (Ian Elliott)
  - Improved the Description section.
  - Added or resolved issues that were found in improving the Description. For example, \texttt{pSurfaceDescription} is used consistently, instead of sometimes using \texttt{pSurface}.
- Revision 4, 2015-05-27 (James Jones)
  - Fixed some grammatical errors and typos
  - Filled in the description of image\texttt{UseFlags} when creating a swapchain.
  - Added a description of swap\texttt{Interval}.
  - Replaced the paragraph describing the order of operations on a queue for image ownership and presentation.
- Revision 5, 2015-05-27 (James Jones)
  - Imported relevant issues from the (abandoned) \texttt{vk_wsi_persistent_swapchain_images} extension.
Added issues 6 and 7, regarding behavior of the acquire next image and present commands with respect to queues.

Updated spec language and examples to align with proposed resolutions to issues 6 and 7.

Revision 6, 2015-05-27 (James Jones)

Added issue 8, regarding atomic presentation of multiple swapchains

Updated spec language and examples to align with proposed resolution to issue 8.

Revision 7, 2015-05-27 (James Jones)

Fixed compilation errors in example code, and made related spec fixes.

Revision 8, 2015-05-27 (James Jones)

Added issue 9, and the related VK_SUBOPTIMAL_KHR result code.

Renamed VK_OUT_OF_DATE_KHR to VK_ERROR_OUT_OF_DATE_KHR.

Revision 9, 2015-05-27 (James Jones)

Added inline proposed resolutions (marked with [JRJ]) to some XXX questions/issues. These should be moved to the issues section in a subsequent update if the proposals are adopted.

Revision 10, 2015-05-28 (James Jones)

Converted vkAcquireNextImageKHR back to a non-queue operation that uses a VkSemaphore object for explicit synchronization.

Added issue 10 to determine whether vkAcquireNextImageKHR generates or returns semaphores, or whether it operates on a semaphore provided by the application.

Revision 11, 2015-05-28 (James Jones)

Marked issues 6, 7, and 8 resolved.

Renamed VkSurfaceCapabilityPropertiesKHR to VkSurfacePropertiesKHR to better convey the mutable nature of the info it contains.

Revision 12, 2015-05-28 (James Jones)

Added issue 11 with a proposed resolution, and the related issue 12.

Updated various sections of the spec to match the proposed resolution to issue 11.

Revision 13, 2015-06-01 (James Jones)

Moved some structures to VK_EXT_KHR_swap_chain to resolve the spec's issues 1 and 2.

Revision 14, 2015-06-01 (James Jones)

Added code for example 4 demonstrating how an application might make use of the two different present and acquire next image KHR result codes.

Added issue 13.

Revision 15, 2015-06-01 (James Jones)

Added issues 14 - 16 and related spec language.

Fixed some spelling errors.

Added language describing the meaningful return values for vkAcquireNextImageKHR and vkQueuePresentKHR.
• Revision 16, 2015-06-02 (James Jones)
  ◦ Added issues 17 and 18, as well as related spec language.
  ◦ Removed some erroneous text added by mistake in the last update.

• Revision 17, 2015-06-15 (Ian Elliott)
  ◦ Changed special value from "-1" to "0" so that the data types can be unsigned.

• Revision 18, 2015-06-15 (Ian Elliott)
  ◦ Clarified the values of VkSurfacePropertiesKHR::minImageCount and the timeout parameter of the vkAcquireNextImageKHR function.

• Revision 19, 2015-06-17 (James Jones)
  ◦ Misc. cleanup. Removed resolved inline issues and fixed typos.
  ◦ Fixed clarification of VkSurfacePropertiesKHR::minImageCount made in version 18.
  ◦ Added a brief "Image Ownership" definition to the list of terms used in the spec.

• Revision 20, 2015-06-17 (James Jones)
  ◦ Updated enum-extending values using new convention.

• Revision 21, 2015-06-17 (James Jones)
  ◦ Added language describing how to use VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR.
  ◦ Cleaned up an XXX comment regarding the description of which queues vkQueuePresentKHR can be used on.

• Revision 22, 2015-06-17 (James Jones)
  ◦ Rebased on Vulkan API version 126.

• Revision 23, 2015-06-18 (James Jones)
  ◦ Updated language for issue 12 to read as a proposed resolution.
  ◦ Marked issues 11, 12, 13, 16, and 17 resolved.
  ◦ Temporarily added links to the relevant bugs under the remaining unresolved issues.
  ◦ Added issues 19 and 20 as well as proposed resolutions.

• Revision 24, 2015-06-19 (Ian Elliott)
  ◦ Changed special value for VkSurfacePropertiesKHR::currentExtent back to "-1" from "0". This value will never need to be unsigned, and "0" is actually a legal value.

• Revision 25, 2015-06-23 (Ian Elliott)
  ◦ Examples now show use of function pointers for extension functions.
  ◦ Eliminated extraneous whitespace.

• Revision 26, 2015-06-25 (Ian Elliott)
  ◦ Resolved Issues 9 & 10 per KHR TSG meeting.

• Revision 27, 2015-06-25 (James Jones)
  ◦ Added oldSwapchain member to VkSwapchainCreateInfoKHR.
• Revision 28, 2015-06-25 (James Jones)
  ◦ Added the “inherit” bits to the rotation and mirroring flags and the associated issue 21.

• Revision 29, 2015-06-25 (James Jones)
  ◦ Added the “clipped” flag to VkSwapchainCreateInfoKHR, and the associated issue 22.
  ◦ Specified that presenting an image does not modify it.

• Revision 30, 2015-06-25 (James Jones)
  ◦ Added language to the spec that clarifies the behavior of vkCreateSwapchainKHR() when the
    oldSwapchain field of VkSwapchainCreateInfoKHR is not NULL.

• Revision 31, 2015-06-26 (Ian Elliott)
  ◦ Example of new VkSwapchainCreateInfoKHR members, “oldSwapchain” and “clipped”.
  ◦ Example of using VkSurfacePropertiesKHR::{min|max}ImageCount to set
    VkSwapchainCreateInfoKHR::minImageCount.
  ◦ Rename vkGetSurfaceInfoKHR()'s 4th parameter to “pDataSize”, for consistency with other
    functions.
  ◦ Add macro with C-string name of extension (just to header file).

• Revision 32, 2015-06-26 (James Jones)
  ◦ Minor adjustments to the language describing the behavior of “oldSwapchain”
  ◦ Fixed the version date on my previous two updates.

• Revision 33, 2015-06-26 (Jesse Hall)
  ◦ Add usage flags to VkSwapchainCreateInfoKHR

• Revision 34, 2015-06-26 (Ian Elliott)
  ◦ Rename vkQueuePresentKHR()'s 2nd parameter to “pPresentInfo”, for consistency with other
    functions.

• Revision 35, 2015-06-26 (Jason Ekstrand)
  ◦ Merged the VkRotationFlagBitsKHR and VkMirrorFlagBitsKHR enums into a single
    VkSurfaceTransformFlagBitsKHR enum.

• Revision 36, 2015-06-26 (Jason Ekstrand)
  ◦ Added a VkSurfaceTransformKHR enum that is not a bitmask. Each value in
    VkSurfaceTransformKHR corresponds directly to one of the bits in
    VkSurfaceTransformFlagBitsKHR so transforming from one to the other is easy. Having a
    separate enum means that currentTransform and preTransform are now unambiguous by
    definition.

• Revision 37, 2015-06-29 (Ian Elliott)
  ◦ Corrected one of the signatures of vkAcquireNextImageKHR, which had the last two
    parameters switched from what it is elsewhere in the specification and header files.

• Revision 38, 2015-06-30 (Ian Elliott)
  ◦ Corrected a typo in description of the vkGetSwapchainInfoKHR() function.
Corrected a typo in header file comment for VkPresentInfoKHR::sType.

Revision 39, 2015-07-07 (Daniel Rakos)
- Added error section describing when each error is expected to be reported.
- Replaced bool32_t with VkBool32.

Revision 40, 2015-07-10 (Ian Elliott)
- Updated to work with version 138 of the vulkan.h header. This includes declaring the VkSwapchainKHR type using the new VK_DEFINE_NONDISP_HANDLE macro, and no longer extending VkObjectFtype (which was eliminated).

Revision 41, 2015-07-09 (Mathias Heyer)
- Added color space language.

Revision 42, 2015-07-10 (Daniel Rakos)
- Updated query mechanism to reflect the convention changes done in the core spec.
- Removed “queue” from the name of VK_STRUCTURE_TYPE_QUEUE_PRESENT_INFO_KHR to be consistent with the established naming convention.
- Removed reference to the no longer existing VkObjectType enum.

Revision 43, 2015-07-17 (Daniel Rakos)
- Added support for concurrent sharing of swapchain images across queue families.
- Updated sample code based on recent changes

Revision 44, 2015-07-27 (Ian Elliott)
- Noted that support for VK_PRESENT_MODE_FIFO_KHR is required. That is ICDs may optionally support IMMEDIATE and MAILBOX, but must support FIFO.

Revision 45, 2015-08-07 (Ian Elliott)
- Corrected a typo in spec file (type and variable name had wrong case for the imageColorSpace member of the VkSwapchainCreateInfoKHR struct).
- Corrected a typo in header file (last parameter in PFN_vkGetSurfacePropertiesKHR was missing “KHR” at the end of type: VkSurfacePropertiesKHR).

Revision 46, 2015-08-20 (Ian Elliott)
- Renamed this extension and all of its enumerations, types, functions, etc. This makes it compliant with the proposed standard for Vulkan extensions.
-Switched from “revision” to “version”, including use of the VK_MAKE_VERSION macro in the header file.
- Made improvements to several descriptions.
- Changed the status of several issues from PROPOSED to RESOLVED, leaving no unresolved issues.
- Resolved several TODOs, did miscellaneous cleanup, etc.

Revision 47, 2015-08-20 (Ian Elliott—porting a 2015-07-29 change from James Jones)
- Moved the surface transform enums to VK_WSI_swapchain so they could be reused by
VK_WSI_display.

- Revision 48, 2015-09-01 (James Jones)
  - Various minor cleanups.

- Revision 49, 2015-09-01 (James Jones)
  - Restore single-field revision number.

- Revision 50, 2015-09-01 (James Jones)
  - Update Example #4 to include code that illustrates how to use the oldSwapchain field.

- Revision 51, 2015-09-01 (James Jones)
  - Fix example code compilation errors.

- Revision 52, 2015-09-08 (Matthaeus G. Chajdas)
  - Corrected a typo.

- Revision 53, 2015-09-10 (Alon Or-bach)
  - Removed underscore from SWAP_CHAIN left in VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR.

- Revision 54, 2015-09-11 (Jesse Hall)
  - Described the execution and memory coherence requirements for image transitions to and from VK_IMAGE_LAYOUT_PRESENT_SOURCE_KHR.

- Revision 55, 2015-09-11 (Ray Smith)
  - Added errors for destroying and binding memory to presentable images

- Revision 56, 2015-09-18 (James Jones)
  - Added fence argument to vkAcquireNextImageKHR
  - Added example of how to meter a host thread based on presentation rate.

- Revision 57, 2015-09-26 (Jesse Hall)
  - Replace VkSurfaceDescriptionKHR with VkSurfaceKHR.
  - Added issue 25 with agreed resolution.

- Revision 58, 2015-09-28 (Jesse Hall)
  - Renamed from VK_EXT_KHR_device_swapchain to VK_EXT_KHR_swapchain.

- Revision 59, 2015-09-29 (Ian Elliott)
  - Changed vkDestroySwapchainKHR() to return void.

- Revision 60, 2015-10-01 (Jeff Vigil)
  - Added error result VK_ERROR_SURFACE_LOST_KHR.

- Revision 61, 2015-10-05 (Jason Ekstrand)
  - Added the VkCompositeAlpha enum and corresponding structure fields.

- Revision 62, 2015-10-12 (Daniel Rakos)
  - Added VK_PRESENT_MODE_FIFO_RELAXED_KHR.
Revision 63, 2015-10-15 (Daniel Rakos)
- Moved surface capability queries to VK_EXT_KHR_surface.

Revision 64, 2015-10-26 (Ian Elliott)
- Renamed from VK_EXT_KHR_swapchain to VK_KHR_swapchain.

Revision 65, 2015-10-28 (Ian Elliott)
- Added optional pResult member to VkPresentInfoKHR, so that per-swapchain results can be obtained from vkQueuePresentKHR().

Revision 66, 2015-11-03 (Daniel Rakos)
- Added allocation callbacks to create and destroy functions.
- Updated resource transition language.
- Updated sample code.

Revision 67, 2015-11-10 (Jesse Hall)
- Add reserved flags bitmask to VkSwapchainCreateInfoKHR.
- Modify naming and member ordering to match API style conventions, and so the VkSwapchainCreateInfoKHR image property members mirror corresponding VkImageCreateInfo members but with an 'image' prefix.
- Make VkPresentInfoKHR::pResults non-const; it is an output array parameter.
- Make pPresentInfo parameter to vkQueuePresentKHR const.

Revision 68, 2016-04-05 (Ian Elliott)
- Moved the “validity” include for vkAcquireNextImage to be in its proper place, after the prototype and list of parameters.
- Clarified language about presentable images, including how they are acquired, when applications can and cannot use them, etc. As part of this, removed language about “ownership” of presentable images, and replaced it with more-consistent language about presentable images being “acquired” by the application.

2016-08-23 (Ian Elliott)
- Update the example code, to use the final API command names, to not have so many characters per line, and to split out a new example to show how to obtain function pointers. This code is more similar to the LunarG “cube” demo program.

2016-08-25 (Ian Elliott)
- A note was added at the beginning of the example code, stating that it will be removed from future versions of the appendix.

Revision 69, 2017-09-07 (Tobias Hector)
- Added interactions with Vulkan 1.1

Revision 70, 2017-10-06 (Ian Elliott)
- Corrected interactions with Vulkan 1.1
VK_KHR_swapchain_mutable_format

Name String

VK_KHR_swapchain_mutable_format

Extension Type

Device extension

Registered Extension Number

201

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_swapchain
• Requires VK_KHR_maintenance2
• Requires VK_KHR_image_format_list

Contact

• Daniel Rakos drakos-arm

Other Extension Metadata

Last Modified Date

2018-03-28

IP Status

No known IP claims.

Contributors

• Jason Ekstrand, Intel
• Jan-Harald Fredriksen, ARM
• Jesse Hall, Google
• Daniel Rakos, AMD
• Ray Smith, ARM

Description

This extension allows processing of swapchain images as different formats to that used by the window system, which is particularly useful for switching between sRGB and linear RGB formats.

It adds a new swapchain creation flag that enables creating image views from presentable images with a different format than the one used to create the swapchain.
New Enum Constants

- VK_KHR_SWAPCHAIN_MUTABLE_FORMAT_EXTENSION_NAME
- VK_KHR_SWAPCHAIN_MUTABLE_FORMAT_SPEC_VERSION
- Extending VkSwapchainCreateFlagBitsKHR:
  - VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR

Issues

1) Are there any new capabilities needed?

RESOLVED: No. It is expected that all implementations exposing this extension support swapchain image format mutability.

2) Do we need a separate VK_SWAPCHAIN_CREATE_EXTENDED_USAGE_BIT_KHR?

RESOLVED: No. This extension requires VK_KHR_maintenance2 and presentable images of swapchains created with VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR are created internally in a way equivalent to specifying both VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT and VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR.

3) Do we need a separate structure to allow specifying an image format list for swapchains?

RESOLVED: No. We simply use the same VkImageFormatListCreateInfoKHR structure introduced by VK_KHR_image_format_list. The structure is required to be included in the pNext chain of VkSwapchainCreateInfoKHR for swapchains created with VK_SWAPCHAIN_CREATE_MUTABLE_FORMAT_BIT_KHR.

Version History

- Revision 1, 2018-03-28 (Daniel Rakos)
  - Internal revisions.

VK_KHR_synchronization2

Name String

VK_KHR_synchronization2

Extension Type

Device extension

Registered Extension Number

315

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
Description

This extension modifies the original core synchronization APIs to simplify the interface and improve usability of these APIs. It also adds new pipeline stage and access flag types that extend into the 64-bit range, as we have run out within the 32-bit range. The new flags are identical to the old values within the 32-bit range, with new stages and bits beyond that.

Pipeline stages and access flags are now specified together in memory barrier structures, making the connection between the two more obvious. Additionally, scoping the pipeline stages into the barrier structs allows the use of the MEMORY_READ and MEMORY_WRITE flags without sacrificing precision. The per-stage access flags should be used to disambiguate specific accesses in a given stage or set of stages - for instance, between uniform reads and sampling operations.

Layout transitions have been simplified as well; rather than requiring a different set of layouts for depth/stencil/color attachments, there are generic VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR and VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR layouts which are contextually applied based on the image format. For example, for a depth format image, VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR is equivalent to VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR. VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR also functionally replaces VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL.

Events are now more efficient, because they include memory dependency information when you set them on the device. Previously, this information was only known when waiting on an event, so the dependencies could not be satisfied until the wait occurred. That sometimes meant stalling the pipeline when the wait occurred. The new API provides enough information for implementations to satisfy these dependencies in parallel with other tasks.

Queue submission has been changed to wrap command buffers and semaphores in extensible structures, which incorporate changes from Vulkan 1.1, VK_KHR_device_group, and VK_KHR_timeline_semaphore. This also adds a pipeline stage to the semaphore signal operation, mirroring the existing pipeline stage specification for wait operations.

Other miscellaneous changes include:
Events can now be specified as interacting only with the device, allowing more efficient access to the underlying object.

Image memory barriers that do not perform an image layout transition can be specified by setting `oldLayout` equal to `newLayout`.

- E.g. the old and new layout can both be set to `VK_IMAGE_LAYOUT_UNDEFINED`, without discarding data in the image.

Queue family ownership transfer parameters are simplified in some cases.

Where two synchronization commands need to be matched up (queue transfer operations, events), the dependency information specified in each place must now match completely for consistency.

Extensions with commands or functions with a `VkPipelineStageFlags` or `VkPipelineStageFlagBits` parameter have had those APIs replaced with equivalents using `VkPipelineStageFlags2KHR`.

The new event and barrier interfaces are now more extensible for future changes.

Relevant pipeline stage masks can now be specified as empty with the new `VK_PIPELINE_STAGE_NONE_KHR` and `VK_PIPELINE_STAGE_2_NONE_KHR` values.

`VkMemoryBarrier2KHR` can be chained to `VkSubpassDependency2`, overriding the original 32-bit stage and access masks.

**New Commands**

- `vkCmdPipelineBarrier2KHR`
- `vkCmdResetEvent2KHR`
- `vkCmdSetEvent2KHR`
- `vkCmdWaitEvents2KHR`
- `vkCmdWriteTimestamp2KHR`
- `vkQueueSubmit2KHR`

If `VK_AMD_buffer_marker` is supported:

- `vkCmdWriteBufferMarker2AMD`

If `VK_NV_device_diagnostic_checkpoints` is supported:

- `vkGetQueueCheckpointData2NV`

**New Structures**

- `VkBufferMemoryBarrier2KHR`
- `VkCommandBufferSubmitInfoKHR`
- `VkDependencyInfoKHR`
- `VkImageMemoryBarrier2KHR`
- `VkSemaphoreSubmitInfoKHR`
- **VkSubmitInfo2KHR**
- Extending **VkPhysicalDeviceFeatures2, VkDeviceCreateInfo**:
  - **VkPhysicalDeviceSynchronization2FeaturesKHR**
- Extending **VkSubpassDependency2**:
  - **VkMemoryBarrier2KHR**

If **VK_NV_device_diagnostic_checkpoints** is supported:
- **VkCheckpointData2NV**
- Extending **VkQueueFamilyProperties2**:
  - **VkQueueFamilyCheckpointProperties2NV**

**New Enums**
- **VkAccessFlagBits2KHR**
- **VkPipelineStageFlagBits2KHR**
- **VkSubmitFlagBitsKHR**

**New Bitmasks**
- **VkAccessFlags2KHR**
- **VkPipelineStageFlags2KHR**
- **VkSubmitFlagsKHR**

**New Enum Constants**
- **VK_KHR_SYNCHRONIZATION_2_EXTENSION_NAME**
- **VK_KHR_SYNCHRONIZATION_2_SPEC_VERSION**
- Extending **VkAccessFlagBits**:
  - **VK_ACCESS_NONE_KHR**
- Extending **VkEventCreateFlagBits**:
  - **VK_EVENT_CREATE_DEVICE_ONLY_BIT_KHR**
- Extending **VkImageLayout**:
  - **VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL_KHR**
  - **VK_IMAGE_LAYOUT_READ_ONLY_OPTIMAL_KHR**
- Extending **VkPipelineStageFlagBits**:
  - **VK_PIPELINE_STAGE_NONE_KHR**
- Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER_2_KHR**
  - **VK_STRUCTURE_TYPE_COMMAND_BUFFER_SUBMIT_INFO_KHR**
If `VK_EXT_blend_operation_advanced` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`

If `VK_EXT_conditional_rendering` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_CONDITIONAL_RENDERING_READ_BIT_EXT`
- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_CONDITIONAL_RENDERING_BIT_EXT`

If `VK_EXT_fragment_density_map` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_FRAGMENT_DENSITY_MAP_READ_BIT_EXT`
- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

If `VK_EXT_transform_feedback` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT`
  - `VK_ACCESS_2_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT`
  - `VK_ACCESS_2_TRANSFORM_FEEDBACK_WRITE_BIT_EXT`
- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_TRANSFORM_FEEDBACK_BIT_EXT`

If `VK_KHR_acceleration_structure` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_KHR`
  - `VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_KHR`
- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_KHR`
If `VK_KHR_fragment_shading_rate` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_FRAGMENT_SHADING_RATE_ATTACHMENT_READ_BIT_KHR`

- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_FRAGMENT_SHADING_RATE_ATTACHMENT_BIT_KHR`

If `VK_KHR_ray_tracing_pipeline` is supported:

- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_KHR`

If `VK_NV_device_diagnostic_checkpoints` is supported:

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_CHECKPOINT_DATA_2_NV`
  - `VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_2_NV`

If `VK_NV_device_generated_commands` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_COMMAND_PREPROCESS_READ_BIT_NV`
  - `VK_ACCESS_2_COMMAND_PREPROCESS_WRITE_BIT_NV`

- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_COMMAND_PREPROCESS_BIT_NV`

If `VK_NV_mesh_shader` is supported:

- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_MESH_SHADER_BIT_NV`
  - `VK_PIPELINE_STAGE_2_TASK_SHADER_BIT_NV`

If `VK_NV_ray_tracing` is supported:

- Extending `VkAccessFlagBits2KHR`:
  - `VK_ACCESS_2_ACCELERATION_STRUCTURE_READ_BIT_NV`
  - `VK_ACCESS_2_ACCELERATION_STRUCTURE_WRITE_BIT_NV`

- Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_ACCELERATION_STRUCTURE_BUILD_BIT_NV`
  - `VK_PIPELINE_STAGE_2_RAY_TRACING_SHADER_BIT_NV`

If `VK_NV_shading_rate_image` is supported:

- Extending `VkAccessFlagBits2KHR`:
Extending `VkPipelineStageFlagBits2KHR`:
  - `VK_PIPELINE_STAGE_2_SHADING_RATE_IMAGE_BIT_NV`

Examples
See [https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples](https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples)

Version History
- Revision 1, 2020-12-03 (Tobias Hector)
  - Internal revisions

**VK_KHR_timeline_semaphore**

**Name String**
- `VK_KHR_timeline_semaphore`

**Extension Type**
- Device extension

**Registered Extension Number**
- 208

**Revision**
- 2

**Extension and Version Dependencies**
- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

**Deprecation state**
- *Promoted* to Vulkan 1.2

**Contact**
- Jason Ekstrand [jekstrand](mailto:jekstrand)

**Other Extension Metadata**

**Last Modified Date**
- 2019-06-12

**IP Status**
- No known IP claims.

**Interactions and External Dependencies**
- This extension interacts with `VK_KHR_external_semaphore_capabilities`
• This extension interacts with `VK_KHR_external_semaphore`
• This extension interacts with `VK_KHR_external_semaphore_win32`
• Promoted to Vulkan 1.2 Core

**Contributors**

• Jeff Bolz, NVIDIA
• Yuriy O'Donnell, Epic Games
• Jason Ekstrand, Intel
• Jesse Hall, Google
• James Jones, NVIDIA
• Jeff Juliano, NVIDIA
• Daniel Rakos, AMD
• Ray Smith, Arm

**Description**

This extension introduces a new type of semaphore that has an integer payload identifying a point in a timeline. Such timeline semaphores support the following operations:

• Host query - A host operation that allows querying the payload of the timeline semaphore.
• Host wait - A host operation that allows a blocking wait for a timeline semaphore to reach a specified value.
• Host signal - A host operation that allows advancing the timeline semaphore to a specified value.
• Device wait - A device operation that allows waiting for a timeline semaphore to reach a specified value.
• Device signal - A device operation that allows advancing the timeline semaphore to a specified value.

**Promotion to Vulkan 1.2**

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

**New Commands**

• `vkGetSemaphoreCounterValueKHR`
• `vkSignalSemaphoreKHR`
• `vkWaitSemaphoresKHR`
New Structures

- VkSemaphoreSignalInfoKHR
- VkSemaphoreWaitInfoKHR
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceTimelineSemaphoreFeaturesKHR
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceTimelineSemaphorePropertiesKHR
- Extending VkSemaphoreCreateInfo, VkPhysicalDeviceExternalSemaphoreInfo:
  - VkSemaphoreTypeCreateInfoKHR
- Extending VkSubmitInfo, VkBindSparseInfo:
  - VkTimelineSemaphoreSubmitInfoKHR

New Enums

- VkSemaphoreTypeKHR
- VkSemaphoreWaitFlagBitsKHR

New Bitmasks

- VkSemaphoreWaitFlagsKHR

New Enum Constants

- VK_KHR_TIMELINE_SEMAPHORE_EXTENSION_NAME
- VK_KHR_TIMELINE_SEMAPHORE_SPEC_VERSION
- Extending VkSemaphoreType:
  - VK_SEMAPHORE_TYPE_BINARY_KHR
  - VK_SEMAPHORE_TYPE_TIMELINE_KHR
- Extending VkSemaphoreWaitFlagBits:
  - VK_SEMAPHORE_WAIT_ANY_BIT_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_FEATURES_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TIMELINE_SEMAPHORE_PROPERTIES_KHR
  - VK_STRUCTURE_TYPE_SEMAPHORE_SIGNAL_INFO_KHR
  - VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO_KHR
  - VK_STRUCTURE_TYPE_SEMAPHORE_WAIT_INFO_KHR
  - VK_STRUCTURE_TYPE_TIMELINE_SEMAPHORE_SUBMIT_INFO_KHR
Issues

1) Do we need a new object type for this?

**RESOLVED:** No, we just introduce a new type of semaphore object, as `VK_KHR_external_semaphore_win32` already uses semaphores as the destination for importing D3D12 fence objects, which are semantically close/identical to the proposed synchronization primitive.

2) What type of payload the new synchronization primitive has?

**RESOLVED:** A 64-bit unsigned integer that can only be set to strictly increasing values by signal operations and is not changed by wait operations.

3) Does the new synchronization primitive have the same signal-before-wait requirement as the existing semaphores do?

**RESOLVED:** No. Timeline semaphores support signaling and waiting entirely asynchronously. It is the responsibility of the client to avoid deadlock.

4) Does the new synchronization primitive allow resetting its payload?

**RESOLVED:** No, allowing the payload value to “go backwards” is problematic. Applications looking for reset behavior should create a new instance of the synchronization primitive instead.

5) How do we enable host waits on the synchronization primitive?

**RESOLVED:** Both a non-blocking query of the current payload value of the synchronization primitive, and a blocking wait operation are provided.

6) How do we enable device waits and signals on the synchronization primitive?

**RESOLVED:** Similar to `VK_KHR_external_semaphore_win32`, this extension introduces a new structure that can be chained to `VkSubmitInfo` to specify the values signaled semaphores should be set to, and the values waited semaphores need to reach.

7) Can the new synchronization primitive be used to synchronize presentation and swapchain image acquisition operations?

**RESOLVED:** Some implementations may have problems with supporting that directly, thus it’s not allowed in this extension.

8) Do we want to support external sharing of the new synchronization primitive type?

**RESOLVED:** Yes. Timeline semaphore specific external sharing capabilities can be queried using `vkGetPhysicalDeviceExternalSemaphoreProperties` by chaining the new `VkSemaphoreTypeCreateInfoKHR` structure to its `pExternalSemaphoreInfo` structure. This allows having a different set of external semaphore handle types supported for timeline semaphores vs binary semaphores.

9) Do we need to add a host signal operation for the new synchronization primitive type?

**RESOLVED:** Yes. This helps in situations where one host thread submits a workload but another
host thread has the information on when the workload is ready to be executed.

10) How should the new synchronization primitive interact with the ordering requirements of the original `VkSemaphore`?

**RESOLVED:** Prior to calling any command which *may* cause a wait operation on a binary semaphore, the client *must* ensure that the semaphore signal operation that has been submitted for execution and any semaphore signal operations on which it depends (if any) *must* have also been submitted for execution.

11) Should we have separate feature bits for different sub-features of timeline semaphores?

**RESOLVED:** No. The only feature which cannot be supported universally is timeline semaphore import/export. For import/export, the client is already required to query available external handle types via `vkGetPhysicalDeviceExternalSemaphoreProperties` and provide the semaphore type by adding a `VkSemaphoreTypeCreateInfoKHR` structure to the `pNext` chain of `VkPhysicalDeviceExternalSemaphoreInfo` so no new feature bit is required.

**Version History**

- Revision 1, 2018-05-10 (Jason Ekstrand)
  - Initial version
- Revision 2, 2019-06-12 (Jason Ekstrand)
  - Added an `initialValue` parameter to timeline semaphore creation

**VK_KHR_uniform_buffer_standard_layout**

**Name String**

`VK_KHR_uniform_buffer_standard_layout`

**Extension Type**

Device extension

**Registered Extension Number**

254

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

**Deprecation state**

- *Promoted* to Vulkan 1.2
Contact
• Graeme Leese

Other Extension Metadata

Last Modified Date
2019-01-25

Interactions and External Dependencies
• Promoted to Vulkan 1.2 Core

Contributors
• Graeme Leese, Broadcom
• Jeff Bolz, NVIDIA
• Tobias Hector, AMD
• Jason Ekstrand, Intel
• Neil Henning, AMD

Description
This extension enables tighter array and struct packing to be used with uniform buffers.

It modifies the alignment rules for uniform buffers, allowing for tighter packing of arrays and structures. This allows, for example, the std430 layout, as defined in GLSL to be supported in uniform buffers.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceUniformBufferStandardLayoutFeaturesKHR

New Enum Constants
• VK_KHR_UNIFORM_BUFFER_STANDARD_LAYOUT_EXTENSION_NAME
• VK_KHR_UNIFORM_BUFFER_STANDARD_LAYOUT_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_UNIFORM_BUFFER_STANDARD_LAYOUT/features_KHR
Version History

• Revision 1, 2019-01-25 (Graeme Leese)
  ◦ Initial draft

VK_KHR_vulkan_memory_model

Name String

VK_KHR_vulkan_memory_model

Extension Type

Device extension

Registered Extension Number

212

Revision

3

Extension and Version Dependencies

• Requires Vulkan 1.0

Deprecation state

• Promoted to Vulkan 1.2

Contact

• Jeff Bolz @jeffbolznv

Other Extension Metadata

Last Modified Date

2018-12-10

IP Status

No known IP claims.

Interactions and External Dependencies

• Promoted to Vulkan 1.2 Core
  • This extension requires SPV_KHR_vulkan_memory_model

Contributors

• Jeff Bolz, NVIDIA
• Alan Baker, Google
• Tobias Hector, AMD
• David Neto, Google
• Robert Simpson, Qualcomm Technologies, Inc.
Description

The VK_KHR_vulkan_memory_model extension allows use of the Vulkan Memory Model, which formally defines how to synchronize memory accesses to the same memory locations performed by multiple shader invocations.

Note

Version 3 of the spec added a member (vulkanMemoryModelAvailabilityVisibilityChains) to VkPhysicalDeviceVulkanMemoryModelFeaturesKHR, which is an incompatible change from version 2.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the KHR suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the vulkanMemoryModel capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceVulkanMemoryModelFeaturesKHR

New Enum Constants

• VK_KHR_VULKAN_MEMORY_MODEL_EXTENSION_NAME
• VK_KHR_VULKAN_MEMORY_MODEL_SPEC_VERSION

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VULKAN_MEMORY_MODEL_FEATURES_KHR

New SPIR-V Capabilities

• VulkanMemoryModelKHR

Version History

• Revision 1, 2018-06-24 (Jeff Bolz)
  ◦ Initial draft
• Revision 3, 2018-12-10 (Jeff Bolz)
  ◦ Add vulkanMemoryModelAvailabilityVisibilityChains member to the VkPhysicalDeviceVulkanMemoryModelFeaturesKHR structure.
VK_KHR_wayland_surface

Name String

VK_KHR_wayland_surface

Extension Type

Instance extension

Registered Extension Number

7

Revision

6

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_surface

Contact

- Jesse Hall @critsec
- Ian Elliott @ianelliottus

Other Extension Metadata

Last Modified Date

2015-11-28

IP Status

No known IP claims.

Contributors

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
- David Mao, AMD
- Norbert Nopper, Freescale
- Alon Or-bach, Samsung
- Daniel Rakos, AMD
Description

The `VK_KHR_wayland_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a Wayland `wl_surface`, as well as a query to determine support for rendering to a Wayland compositor.

New Commands

- `vkCreateWaylandSurfaceKHR`
- `vkGetPhysicalDeviceWaylandPresentationSupportKHR`

New Structures

- `VkWaylandSurfaceCreateInfoKHR`

New Bitmasks

- `VkWaylandSurfaceCreateFlagsKHR`

New Enum Constants

- `VK_KHR_WAYLAND_SURFACE_EXTENSION_NAME`
- `VK_KHR_WAYLAND_SURFACE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_WAYLAND_SURFACE_CREATE_INFO_KHR`

Issues

1) Does Wayland need a way to query for compatibility between a particular physical device and a specific Wayland display? This would be a more general query than `vkGetPhysicalDeviceSurfaceSupportKHR`: if the Wayland-specific query returned `VK_TRUE` for a `(VkPhysicalDevice, struct wl_display*)` pair, then the physical device could be assumed to support presentation to any `VkSurfaceKHR` for surfaces on the display.

**RESOLVED:** Yes. `vkGetPhysicalDeviceWaylandPresentationSupportKHR` was added to address this issue.

2) Should we require surfaces created with `vkCreateWaylandSurfaceKHR` to support the `VK_PRESENT_MODE_MAILBOX_KHR` present mode?

**RESOLVED:** Yes. Wayland is an inherently mailbox window system and mailbox support is
required for some Wayland compositor interactions to work as expected. While handling these interactions may be possible with \texttt{VK\_PRESENT\_MODE\_FIFO\_KHR}, it is much more difficult to do without deadlock and requiring all Wayland applications to be able to support implementations which only support \texttt{VK\_PRESENT\_MODE\_FIFO\_KHR} would be an onerous restriction on application developers.

**Version History**

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft, based on the previous contents of \texttt{VK\_EXT\_KHR\_swapchain} (later renamed \texttt{VK\_EXT\_KHR\_surface}).
- Revision 2, 2015-10-02 (James Jones)
  - Added \texttt{vkGetPhysicalDeviceWaylandPresentationSupportKHR()} to resolve issue #1.
  - Adjusted wording of issue #1 to match the agreed-upon solution.
  - Renamed “window” parameters to “surface” to match Wayland conventions.
- Revision 3, 2015-10-26 (Ian Elliott)
  - Renamed from \texttt{VK\_EXT\_KHR\_wayland\_surface} to \texttt{VK\_KHR\_wayland\_surface}.
- Revision 4, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to \texttt{vkCreateWaylandSurfaceKHR}.
- Revision 5, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a \texttt{pCreateInfo} structure.
- Revision 6, 2017-02-08 (Jason Ekstrand)
  - Added the requirement that implementations support \texttt{VK\_PRESENT\_MODE\_MAILBOX\_KHR}.
  - Added wording about interactions between \texttt{vkQueuePresentKHR} and the Wayland requests sent to the compositor.

**VK\_KHR\_win32\_keyed\_mutex**

**Name String**

\texttt{VK\_KHR\_win32\_keyed\_mutex}

**Extension Type**

Device extension

**Registered Extension Number**

76

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires \texttt{VK\_KHR\_external\_memory\_win32}
Description

Applications that wish to import Direct3D 11 memory objects into the Vulkan API may wish to use the native keyed mutex mechanism to synchronize access to the memory between Vulkan and Direct3D. This extension provides a way for an application to access the keyed mutex associated with an imported Vulkan memory object when submitting command buffers to a queue.

New Structures

- Extending \texttt{VkSubmitInfo, VkSubmitInfo2KHR}:
  - \texttt{VkWin32KeyedMutexAcquireReleaseInfoKHR}

New Enum Constants

- \texttt{VK_KHR_WIN32_KEYED_MUTEX_EXTENSION_NAME}
- \texttt{VK_KHR_WIN32_KEYED_MUTEX_SPEC_VERSION}
- Extending \texttt{VkStructureType}:
  - \texttt{VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_KHR}

Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

VK_KHR_win32_surface

Name String

\texttt{VK_KHR_win32_surface}
Extension Type
  Instance extension

Registered Extension Number
  10

Revision
  6

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_surface

Contact
  • Jesse Hall @critsec
  • Ian Elliott @ianelliottus

Other Extension Metadata

Last Modified Date
  2017-04-24

IP Status
  No known IP claims.

Contributors
  • Patrick Doane, Blizzard
  • Jason Ekstrand, Intel
  • Ian Elliott, LunarG
  • Courtney Goeltzenleuchter, LunarG
  • Jesse Hall, Google
  • James Jones, NVIDIA
  • Antoine Labour, Google
  • Jon Leech, Khronos
  • David Mao, AMD
  • Norbert Nopper, Freescale
  • Alon Or-bach, Samsung
  • Daniel Rakos, AMD
  • Graham Sellers, AMD
  • Ray Smith, ARM
  • Jeff Vigil, Qualcomm
Description

The `VK_KHR_win32_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a Win32 `HWND`, as well as a query to determine support for rendering to the windows desktop.

New Commands

- `vkCreateWin32SurfaceKHR`
- `vkGetPhysicalDeviceWin32PresentationSupportKHR`

New Structures

- `VkWin32SurfaceCreateInfoKHR`

New Bitmasks

- `VkWin32SurfaceCreateFlagsKHR`

New Enum Constants

- `VK_KHR_WIN32_SURFACE_EXTENSION_NAME`
- `VK_KHR_WIN32_SURFACE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR`

Issues

1) Does Win32 need a way to query for compatibility between a particular physical device and a specific screen? Compatibility between a physical device and a window generally only depends on what screen the window is on. However, there is not an obvious way to identify a screen without already having a window on the screen.

**RESOLVED:** No. While it may be useful, there is not a clear way to do this on Win32. However, a method was added to query support for presenting to the window desktop as a whole.

2) If a native window object (`HWND`) is used by one graphics API, and then is later used by a different graphics API (one of which is Vulkan), can these uses interfere with each other?

**RESOLVED:** Yes.

Uses of a window object by multiple graphics APIs results in undefined behavior. Such behavior may succeed when using one Vulkan implementation but fail when using a different Vulkan implementation. Potential failures include:

- Creating then destroying a flip presentation model DXGI swapchain on a window object can
prevent `vkCreateSwapchainKHR` from succeeding on the same window object.

- Creating then destroying a `VkSwapchainKHR` on a window object can prevent creation of a bitblt model DXGI swapchain on the same window object.

- Creating then destroying a `VkSwapchainKHR` on a window object can effectively `SetPixelFormat` to a different format than the format chosen by an OpenGL application.

- Creating then destroying a `VkSwapchainKHR` on a window object on one `VkPhysicalDevice` can prevent `vkCreateSwapchainKHR` from succeeding on the same window object, but on a different `VkPhysicalDevice` that is associated with a different Vulkan ICD.

In all cases the problem can be worked around by creating a new window object.

Technical details include:

- Creating a DXGI swapchain over a window object can alter the object for the remainder of its lifetime. The alteration persists even after the DXGI swapchain has been destroyed. This alteration can make it impossible for a conformant Vulkan implementation to create a `VkSwapchainKHR` over the same window object. Mention of this alteration can be found in the remarks section of the MSDN documentation for `DXGI_SWAP_EFFECT`.

- Calling GDI's `SetPixelFormat` (needed by OpenGL's WGL layer) on a window object alters the object for the remainder of its lifetime. The MSDN documentation for `SetPixelFormat` explains that a window object's pixel format can be set only one time.

- Creating a `VkSwapchainKHR` over a window object can alter the object for its remaining lifetime. Either of the above alterations may occur as a side effect of `vkCreateSwapchainKHR`.

**Version History**

- Revision 1, 2015-09-23 (Jesse Hall)
  - Initial draft, based on the previous contents of VK_EXT_KHR_swapchain (later renamed VK_EXT_KHR_surface).

- Revision 2, 2015-10-02 (James Jones)
  - Added presentation support query for win32 desktops.

- Revision 3, 2015-10-26 (Ian Elliott)
  - Renamed from VK_EXT_KHR_win32_surface to VK_KHR_win32_surface.

- Revision 4, 2015-11-03 (Daniel Rakos)
  - Added allocation callbacks to `vkCreateWin32SurfaceKHR`.

- Revision 5, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a `pCreateInfo` structure.

- Revision 6, 2017-04-24 (Jeff Juliano)
  - Add issue 2 addressing reuse of a native window object in a different Graphics API, or by a different Vulkan ICD.
VK_KHR_workgroup_memory_explicit_layout

Name String
VK_KHR_workgroup_memory_explicit_layout

Extension Type
Device extension

Registered Extension Number
337

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact
- Caio Marcelo de Oliveira Filho

Other Extension Metadata

Last Modified Date
2020-06-01

IP Status
No known IP claims.

Interactions and External Dependencies
- This extension requires SPV_KHR_workgroup_memory_explicit_layout

Contributors
- Caio Marcelo de Oliveira Filho, Intel
- Jeff Bolz, NVIDIA
- Graeme Leese, Broadcom
- Jason Ekstrand, Intel
- Daniel Koch, NVIDIA

Description
This extension adds Vulkan support for the SPV_KHR_workgroup_memory_explicit_layout SPIR-V extension, which allows shaders to explicitly define the layout of Workgroup storage class memory and create aliases between variables from that storage class in a compute shader.

The aliasing feature allows different “views” on the same data, so the shader can bulk copy data
from another storage class using one type (e.g. an array of large vectors), and then use the data with a more specific type. It also enables reducing the amount of workgroup memory consumed by allowing the shader to alias data whose lifetimes don't overlap.

The explicit layout support and some form of aliasing is also required for layering OpenCL on top of Vulkan.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceWorkgroupMemoryExplicitLayoutFeaturesKHR`

**New Enum Constants**

- `VK_KHR_WORKGROUP_MEMORY_EXPLICIT_LAYOUT_EXTENSION_NAME`
- `VK_KHR_WORKGROUP_MEMORY_EXPLICIT_LAYOUT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_WORKGROUP_MEMORY_EXPLICIT_LAYOUT/features_KHR`

**New SPIR-V Capabilities**

- `WorkgroupMemoryExplicitLayoutKHR`
- `WorkgroupMemoryExplicitLayout8BitAccessKHR`
- `WorkgroupMemoryExplicitLayout16BitAccessKHR`

**Version History**

- Revision 1, 2020-06-01 (Caio Marcelo de Oliveira Filho)
  - Initial version

**VK_KHR_xcb_surface**

**Name String**

`VK_KHR_xcb_surface`

**Extension Type**

Instance extension

**Registered Extension Number**

6

**Revision**

6

**Extension and Version Dependencies**

- Requires Vulkan 1.0
Description

The `VK_KHR_xcb_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to an X11 `Window`, using the XCB client-side library, as well as a query to determine support for rendering via XCB.

New Commands

- `vkCreateXcbSurfaceKHR`
• vkGetPhysicalDeviceXcbPresentationSupportKHR

New Structures
• VkXcbSurfaceCreateInfoKHR

New Bitmasks
• VkXcbSurfaceCreateFlagsKHR

New Enum Constants
• VK_KHR_XCB_SURFACE_EXTENSION_NAME
• VK_KHR_XCB_SURFACE_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR

Issues

1) Does XCB need a way to query for compatibility between a particular physical device and a specific screen? This would be a more general query than vkGetPhysicalDeviceSurfaceSupportKHR: If it returned VK_TRUE, then the physical device could be assumed to support presentation to any window on that screen.

RESOLVED: Yes, this is needed for toolkits that want to create a VkDevice before creating a window. To ensure the query is reliable, it must be made against a particular X visual rather than the screen in general.

Version History

• Revision 1, 2015-09-23 (Jesse Hall)
  ◦ Initial draft, based on the previous contents of VK_EXT_KHR_swapchain (later renamed VK_EXT_KHR_surface).

• Revision 2, 2015-10-02 (James Jones)
  ◦ Added presentation support query for an (xcb_connection_t*, xcb_visualid_t) pair.
  ◦ Removed “root” parameter from CreateXcbSurfaceKHR(), as it is redundant when a window on the same screen is specified as well.
  ◦ Adjusted wording of issue #1 and added agreed upon resolution.

• Revision 3, 2015-10-14 (Ian Elliott)
  ◦ Removed “root” parameter from CreateXcbSurfaceKHR() in one more place.

• Revision 4, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_xcb_surface to VK_KHR_xcb_surface.

• Revision 5, 2015-10-23 (Daniel Rakos)
Added allocation callbacks to `vkCreateXcbSurfaceKHR`.

- Revision 6, 2015-11-28 (Daniel Rakos)
  - Updated the surface create function to take a `pCreateInfo` structure.

**VK_KHR_xlib_surface**

**Name String**

```
VK_KHR_xlib_surface
```

**Extension Type**

- Instance extension

**Registered Extension Number**

5

**Revision**

6

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

**Contact**

- Jesse Hall (@critsec)
- Ian Elliott (@ianelliottus)

**Other Extension Metadata**

**Last Modified Date**

2015-11-28

**IP Status**

No known IP claims.

**Contributors**

- Patrick Doane, Blizzard
- Jason Ekstrand, Intel
- Ian Elliott, LunarG
- Courtney Goeltzenleuchter, LunarG
- Jesse Hall, Google
- James Jones, NVIDIA
- Antoine Labour, Google
- Jon Leech, Khronos
Description

The `VK_KHR_xlib_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to an X11 Window, using the Xlib client-side library, as well as a query to determine support for rendering via Xlib.

New Commands

- `vkCreateXlibSurfaceKHR`
- `vkGetPhysicalDeviceXlibPresentationSupportKHR`

New Structures

- `VkXlibSurfaceCreateInfoKHR`

New Bitmasks

- `VkXlibSurfaceCreateFlagsKHR`

New Enum Constants

- `VK_KHR_XLIB_SURFACE_EXTENSION_NAME`
- `VK_KHR_XLIB_SURFACE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_XLIB_SURFACE_CREATE_INFO_KHR`

Issues

1) Does X11 need a way to query for compatibility between a particular physical device and a specific screen? This would be a more general query than `vkGetPhysicalDeviceSurfaceSupportKHR`; if it returned `VK_TRUE`, then the physical device could be assumed to support presentation to any window on that screen.

**RESOLVED:** Yes, this is needed for toolkits that want to create a `VkDevice` before creating a window. To ensure the query is reliable, it must be made against a particular X visual rather than the screen.
in general.

Version History

- Revision 1, 2015-09-23 (Jesse Hall)
  ◦ Initial draft, based on the previous contents of VK_EXT_KHR_swapchain (later renamed VK_EXT_KHR_surface).
- Revision 2, 2015-10-02 (James Jones)
  ◦ Added presentation support query for (Display*, VisualID) pair.
  ◦ Removed “root” parameter from CreateXlibSurfaceKHR(), as it is redundant when a window on the same screen is specified as well.
  ◦ Added appropriate X errors.
  ◦ Adjusted wording of issue #1 and added agreed upon resolution.
- Revision 3, 2015-10-14 (Ian Elliott)
  ◦ Renamed this extension from VK_EXT_KHR_x11_surface to VK_EXT_KHR_xlib_surface.
- Revision 4, 2015-10-26 (Ian Elliott)
  ◦ Renamed from VK_EXT_KHR_xlib_surface to VK_KHR_xlib_surface.
- Revision 5, 2015-11-03 (Daniel Rakos)
  ◦ Added allocation callbacks to vkCreateXlibSurfaceKHR.
- Revision 6, 2015-11-28 (Daniel Rakos)
  ◦ Updated the surface create function to take a pCreateInfo structure.

VK_KHR_zero_initialize_workgroup_memory

Name String

VK_KHR_zero_initialize_workgroup_memory

Extension Type

Device extension

Registered Extension Number

326

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Alan Baker alan-baker
This extension allows the use of a null constant initializer on shader Workgroup memory variables, allowing implementations to expose any special hardware or instructions they may have. Zero initialization is commonly used by applications running untrusted content (e.g. web browsers) as a way of defeating memory-scraping attacks.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceZeroInitializeWorkgroupMemoryFeaturesKHR`

**New Enum Constants**

- `VK_KHR_ZERO_INITIALIZE_WORKGROUP_MEMORY_EXTENSION_NAME`
- `VK_KHR_ZERO_INITIALIZE_WORKGROUP_MEMORY_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ZERO_INITIALIZE_WORKGROUP_MEMORY_FEATURES_KHR`

**Version History**

- Revision 1, 2020-11-18 (Alan Baker)
  - Internal draft version

**VK_EXT_4444Formats**

**Name String**

`VK_EXT_4444_formats`

**Extension Type**

Device extension
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Joshua Ashton

Other Extension Metadata

Last Modified Date
2020-07-28

IP Status
No known IP claims.

Contributors

- Joshua Ashton, Valve
- Jason Ekstrand, Intel

Description

This extension defines the `VK_FORMAT_A4R4G4B4_UNORM_PACK16_EXT` and `VK_FORMAT_A4B4G4R4_UNORM_PACK16_EXT` formats which are defined in other current graphics APIs.

This extension may be useful for building translation layers for those APIs or for porting applications that use these formats without having to resort to swizzles.

When `VK_EXT_custom_border_color` is used, these formats are not subject to the same restrictions for border color without format as with `VK_FORMAT_B4G4R4A4_UNORM_PACK16`.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDevice4444FormatsFeaturesEXT`

New Enum Constants

- `VK_EXT_4444_FORMATS_EXTENSION_NAME`
- `VK_EXT_4444_FORMATS_SPEC_VERSION`
- Extending `VkFormat`: 
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_4444_FORMATS_FEATURES_EXT

Version History

• Revision 1, 2020-07-04 (Joshua Ashton)
  ◦ Initial draft

VK_EXT_acquire_xlib_display

Name String

  VK_EXT_acquire_xlib_display

Extension Type

  Instance extension

Registered Extension Number

  90

Revision

  1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_EXT_direct_mode_display

Contact

• James Jones cubanismo

Other Extension Metadata

Last Modified Date

  2016-12-13

IP Status

  No known IP claims.

Contributors

• Dave Airlie, Red Hat
• Pierre Boudier, NVIDIA
• James Jones, NVIDIA
• Damien Leone, NVIDIA
Description

This extension allows an application to take exclusive control on a display currently associated with
an X11 screen. When control is acquired, the display will be deassociated from the X11 screen until
control is released or the specified display connection is closed. Essentially, the X11 screen will
behave as if the monitor has been unplugged until control is released.

New Commands

- vkAcquireXlibDisplayEXT
- vkGetRandROutputDisplayEXT

New Enum Constants

- VK_EXT_ACQUIRE_XLIB_DISPLAY_EXTENSION_NAME
- VK_EXT_ACQUIRE_XLIB_DISPLAY_SPEC_VERSION

Issues

1) Should vkAcquireXlibDisplayEXT take an RandR display ID, or a Vulkan display handle as input?

RESOLVED: A Vulkan display handle. Otherwise there would be no way to specify handles to
displays that had been prevented from being included in the X11 display list by some native
platform or vendor-specific mechanism.

2) How does an application figure out which RandR display corresponds to a Vulkan display?

RESOLVED: A new function, vkGetRandROutputDisplayEXT, is introduced for this purpose.

3) Should vkGetRandROutputDisplayEXT be part of this extension, or a general Vulkan / RandR or
Vulkan / Xlib extension?

RESOLVED: To avoid yet another extension, include it in this extension.

Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

VK_EXT_astc_decode_mode

Name String

  VK_EXT_astc_decode_mode
Extension Type
    Device extension

Registered Extension Number
    68

Revision
    1

Extension and Version Dependencies
    • Requires Vulkan 1.0
    • Requires VK_KHR_get_physical_device_properties2

Contact
    • Jan-Harald Fredriksen janharaldfredriksen-arm

Other Extension Metadata

Last Modified Date
    2018-08-07

Contributors
    • Jan-Harald Fredriksen, Arm

Description

The existing specification requires that low dynamic range (LDR) ASTC textures are decompressed to FP16 values per component. In many cases, decompressing LDR textures to a lower precision intermediate result gives acceptable image quality. Source material for LDR textures is typically authored as 8-bit UNORM values, so decoding to FP16 values adds little value. On the other hand, reducing precision of the decoded result reduces the size of the decompressed data, potentially improving texture cache performance and saving power.

The goal of this extension is to enable this efficiency gain on existing ASTC texture data. This is achieved by giving the application the ability to select the intermediate decoding precision.

Three decoding options are provided:

• Decode to VK_FORMAT_R16G16B16A16_SFLOAT precision: This is the default, and matches the required behavior in the core API.

• Decode to VK_FORMAT_R8G8B8A8_UNORM precision: This is provided as an option in LDR mode.

• Decode to VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 precision: This is provided as an option in both LDR and HDR mode. In this mode, negative values cannot be represented and are clamped to zero. The alpha component is ignored, and the results are as if alpha was 1.0. This decode mode is optional and support can be queried via the physical device properties.
New Structures

- Extending `VkImageViewCreateInfo`:
  - `VkImageViewASTCDecodeModeEXT`

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceASTCDecodeFeaturesEXT`

New Enum Constants

- `VK_EXT_ASTC_DECODE_MODE_EXTENSION_NAME`
- `VK_EXT_ASTC_DECODE_MODE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ASTC_DECODE_FEATURES_EXT`

Issues

1) Are implementations allowed to decode at a higher precision than what is requested?

RESOLUTION: No.
If we allow this, then this extension could be exposed on all implementations that support ASTC. But developers would have no way of knowing what precision was actually used, and thus whether the image quality is sufficient at reduced precision.

2) Should the decode mode be image view state and/or sampler state?

RESOLUTION: Image view state only.
Some implementations treat the different decode modes as different texture formats.

Example

Create an image view that decodes to `VK_FORMAT_R8G8B8A8_UNORM` precision:
VkImageViewASTCDecodeModeEXT decodeMode =
{
    VK_STRUCTURE_TYPE_IMAGE_VIEW_ASTC_DECODE_MODE_EXT, // sType
    NULL, // pNext
    VK_FORMAT_R8G8B8A8_UNORM // decode mode
};

VkImageViewCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO, // sType
    &decodeMode, // pNext
    // flags, image, viewType set to application-desired values
    VK_FORMAT_ASTC_8x8_UNORM_BLOCK, // format
    // components, subresourceRange set to application-desired values
};

VkImageView imageView;
VkResult result = vkCreateImageView(
    device,
    &createInfo,
    NULL,
    &imageView);

Version History

• Revision 1, 2018-08-07 (Jan-Harald Fredriksen)
  ◦ Initial revision

VK_EXT_blend_operation_advanced

Name String

VK_EXT_blend_operation_advanced

Extension Type

Device extension

Registered Extension Number

149

Revision

2

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Jeff Bolz (@jeffbolznv)
Description

This extension adds a number of “advanced” blending operations that can be used to perform new color blending operations, many of which are more complex than the standard blend modes provided by unextended Vulkan. This extension requires different styles of usage, depending on the level of hardware support and the enabled features:

- If `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT::advancedBlendCoherentOperations` is `VK_FALSE`, the new blending operations are supported, but a memory dependency must separate each advanced blend operation on a given sample. `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT` is used to synchronize reads using advanced blend operations.

- If `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT::advancedBlendCoherentOperations` is `VK_TRUE`, advanced blend operations obey primitive order just like basic blend operations.

In unextended Vulkan, the set of blending operations is limited, and can be expressed very simply. The `VK_BLEND_OP_MIN` and `VK_BLEND_OP_MAX` blend operations simply compute component-wise minimums or maximums of source and destination color components. The `VK_BLEND_OP_ADD`, `VK_BLEND_OP_SUBTRACT`, and `VK_BLEND_OP_REVERSE_SUBTRACT` modes multiply the source and destination colors by source and destination factors and either add the two products together or subtract one from the other. This limited set of operations supports many common blending operations but precludes the use of more sophisticated transparency and blending operations commonly available in many dedicated imaging APIs.

This extension provides a number of new “advanced” blending operations. Unlike traditional blending operations using `VK_BLEND_OP_ADD`, these blending equations do not use source and destination factors specified by `VkBlendFactor`. Instead, each blend operation specifies a complete equation based on the source and destination colors. These new blend operations are used for both RGB and alpha components; they must not be used to perform separate RGB and alpha blending (via different values of color and alpha `VkBlendOp`).

These blending operations are performed using premultiplied colors, where RGB colors can be considered premultiplied or non-premultiplied by alpha, according to the `srcPremultiplied` and `dstPremultiplied` members of `VkPipelineColorBlendAdvancedStateCreateInfoEXT`. If a color is considered non-premultiplied, the (R,G,B) color components are multiplied by the alpha component prior to blending. For non-premultiplied color components in the range [0,1], the corresponding premultiplied color component would have values in the range [0 × A, 1 × A].

Many of these advanced blending equations are formulated where the result of blending source and destination colors with partial coverage have three separate contributions: from the portions covered by both the source and the destination, from the portion covered only by the source, and...
from the portion covered only by the destination. The blend parameter
`VkPipelineColorBlendAdvancedStateCreateInfoEXT:blendOverlap` can be used to specify a
correlation between source and destination pixel coverage. If set to `VK_BLEND_OVERLAP_CONJOINT_EXT`,
the source and destination are considered to have maximal overlap, as would be the case if
drawing two objects on top of each other. If set to `VK_BLEND_OVERLAP_DISJOINT_EXT`, the source and
destination are considered to have minimal overlap, as would be the case when rendering a
complex polygon tessellated into individual non-intersecting triangles. If set to
`VK_BLEND_OVERLAP_UNCORRELATED_EXT`, the source and destination coverage are assumed to have no
spatial correlation within the pixel.

In addition to the coherency issues on implementations not supporting
`advancedBlendCoherentOperations`, this extension has several limitations worth noting. First, the new
blend operations have a limit on the number of color attachments they can be used with, as
indicated by `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT::advancedBlendMaxColorAttachments`. Additionally, blending precision may be limited to 16-bit
floating-point, which may result in a loss of precision and dynamic range for framebuffer formats
with 32-bit floating-point components, and in a loss of precision for formats with 12- and 16-bit
signed or unsigned normalized integer components.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceBlendOperationAdvancedFeaturesEXT`

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceBlendOperationAdvancedPropertiesEXT`

- Extending `VkPipelineColorBlendStateCreateInfo`:
  - `VkPipelineColorBlendAdvancedStateCreateInfoEXT`

**New Enums**

- `VkBlendOverlapEXT`

**New Enum Constants**

- `VK_EXT_BLEND_OPERATION_ADVANCED_EXTENSION_NAME`
- `VK_EXT_BLEND_OPERATION_ADVANCED_SPEC_VERSION`

- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_COLOR_ATTACHMENT_READ_NONCOHERENT_BIT_EXT`

- Extending `VkBlendOp`:
  - `VK_BLEND_OP_BLUE_EXT`
  - `VK_BLEND_OP_COLORBURN_EXT`
  - `VK_BLEND_OP_COLORDODGE_EXT`
  - `VK_BLEND_OP_CONTRAST_EXT`
- VK_BLEND_OP_DARKEN_EXT
- VK_BLEND_OP_DIFFERENCE_EXT
- VK_BLEND_OP_DST_ATOP_EXT
- VK_BLEND_OP_DST_EXT
- VK_BLEND_OP_DST_IN_EXT
- VK_BLEND_OP_DST_OUT_EXT
- VK_BLEND_OP_DST_OVER_EXT
- VK_BLEND_OP_EXCLUSION_EXT
- VK_BLEND_OP_GREEN_EXT
- VK_BLEND_OP_HARDLIGHT_EXT
- VK_BLEND_OP_HARDMIX_EXT
- VK_BLEND_OP_HSL_COLOR_EXT
- VK_BLEND_OP_HSL_HUE_EXT
- VK_BLEND_OP_HSL_LUMINOSITY_EXT
- VK_BLEND_OP_HSL_SATURATION_EXT
- VK_BLEND_OP_INVERT_EXT
- VK_BLEND_OP_INVERT_OVG_EXT
- VK_BLEND_OP_INVERT_RGB_EXT
- VK_BLEND_OP_LIGHTEN_EXT
- VK_BLEND_OP_LINEARBURN_EXT
- VK_BLEND_OP_LINEARDODGE_EXT
- VK_BLEND_OP_LINEARLIGHT_EXT
- VK_BLEND_OP_MINUS_CLAMPED_EXT
- VK_BLEND_OP_MINUS_EXT
- VK_BLEND_OP_MULTIPLY_EXT
- VK_BLEND_OP_OVERLAY_EXT
- VK_BLEND_OP_PINLIGHT_EXT
- VK_BLEND_OP_PLUS_CLAMPED_ALPHA_EXT
- VK_BLEND_OP_PLUS_CLAMPED_EXT
- VK_BLEND_OP_PLUS_DARKER_EXT
- VK_BLEND_OP_PLUS_EXT
- VK_BLEND_OP_RED_EXT
- VK_BLEND_OP_SCREEN_EXT
- VK_BLEND_OP_SOFTLIGHT_EXT
- VK_BLEND_OP_SRC_ATOP_EXT
VK_BLEND_OP_SRC_EXT
VK_BLEND_OP_SRC_IN_EXT
VK_BLEND_OP_SRC_OUT_EXT
VK_BLEND_OP_SRC_OVER_EXT
VK_BLEND_OP_VIVIDLIGHT_EXT
VK_BLEND_OP_XOR_EXT
VK_BLEND_OP_ZERO_EXT

• Extending VkStructureType:
  • VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_FEATURES_EXT
  • VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BLEND_OPERATION_ADVANCED_PROPERTIES_EXT
  • VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ADVANCED_STATE_CREATE_INFO_EXT

Issues
None.

Version History

• Revision 1, 2017-06-12 (Jeff Bolz)
  • Internal revisions
• Revision 2, 2017-06-12 (Jeff Bolz)
  • Internal revisions

VK_EXT_calibrated_timestamps

Name String
VK_EXT_calibrated_timestamps

Extension Type
Device extension

Registered Extension Number
185

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Daniel Rakos drakos-amd
Description

This extension provides an interface to query calibrated timestamps obtained quasi simultaneously from two time domains.

New Commands

- vkGetCalibratedTimestampsEXT
- vkGetPhysicalDeviceCalibrateableTimeDomainsEXT

New Structures

- VkCalibratedTimestampInfoEXT

New Enums

- VkTimeDomainEXT

New Enum Constants

- VK_EXT_CALIBRATED_TIMESTAMPS_EXTENSION_NAME
- VK_EXT_CALIBRATED_TIMESTAMPS_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_CALIBRATED_TIMESTAMP_INFO_EXT

Issues

1) Is the device timestamp value returned in the same time domain as the timestamp values written by vkCmdWriteTimestamp?
RESOLVED: Yes.

2) What time domain is the host timestamp returned in?

RESOLVED: A query is provided to determine the calibrateable time domains. The expected host time domain used on Windows is that of QueryPerformanceCounter, and on Linux that of CLOCK_MONOTONIC.

3) Should we support other time domain combinations than just one host and the device time domain?

RESOLVED: Supporting that would need the application to query the set of supported time domains, while supporting only one host and the device time domain would only need a query for the host time domain type. The proposed API chooses the general approach for the sake of extensibility.

4) Shouldn’t we use CLOCK_MONOTONIC_RAW instead of CLOCK_MONOTONIC?

RESOLVED: CLOCK_MONOTONIC is usable in a wider set of situations, however, it is subject to NTP adjustments so some use cases may prefer CLOCK_MONOTONIC_RAW. Thus this extension allows both to be exposed.

5) How can the application extrapolate future device timestamp values from the calibrated timestamp value?

RESOLVED: VkPhysicalDeviceLimits::timestampPeriod makes it possible to calculate future device timestamps as follows:

\[
\text{futureTimestamp} = \frac{\text{calibratedTimestamp} + \text{deltaNanoseconds}}{\text{timestampPeriod}}
\]

6) In what queue are timestamp values in time domain VK_TIME_DOMAIN_DEVICE_EXT captured by vkGetCalibratedTimestampsEXT?

RESOLVED: An implementation supporting this extension will have all its VkQueue share the same time domain.

6) Can the host and device timestamp values drift apart over longer periods of time?

RESOLVED: Yes, especially as some time domains by definition allow for that to happen (e.g. CLOCK_MONOTONIC is subject to NTP adjustments). Thus it's recommended that applications re-calibrate from time to time.

7) Should we add a query for reporting the maximum deviation of the timestamp values returned by calibrated timestamp queries?

RESOLVED: A global query seems inappropriate and difficult to enforce. However, it’s possible to return the maximum deviation any single calibrated timestamp query can have by sampling one of the time domains twice as follows:
8) Can the maximum deviation reported ever be zero?

**RESOLVED**: Unless the tick of each clock corresponding to the set of time domains coincides and all clocks can literally be sampled simultaneously, there isn’t really a possibility for the maximum deviation to be zero, so by convention the maximum deviation is always at least the maximum of the length of the ticks of the set of time domains calibrated and thus can never be zero.

**Version History**

- Revision 2, 2021-03-16 (Lionel Landwerlin)
  - Specify requirement on device timestamps
- Revision 1, 2018-10-04 (Daniel Rakos)
  - Internal revisions.

**VK_EXT_color_write_enable**

**Name String**

VK_EXT_color_write_enable

**Extension Type**

Device extension

**Registered Extension Number**

382

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Sharif Elcott 🌐selcott

**Other Extension Metadata**

**Last Modified Date**

2020-02-25
IP Status
No known IP claims.

Contributors
• Sharif Elcott, Google
• Tobias Hector, AMD
• Piers Daniell, NVIDIA

Description
This extension allows for selectively enabling and disabling writes to output color attachments via a pipeline dynamic state.

The intended use cases for this new state are mostly identical to those of colorWriteMask, such as selectively disabling writes to avoid feedback loops between subpasses or bandwidth savings for unused outputs. By making the state dynamic, one additional benefit is the ability to reduce pipeline counts and pipeline switching via shaders that write a superset of the desired data of which subsets are selected dynamically. The reason for a new state, colorWriteEnable, rather than making colorWriteMask dynamic is that, on many implementations, the more flexible per-channel semantics of the colorWriteMask state cannot be made dynamic in a performant manner.

New Commands
• vkCmdSetColorWriteEnableEXT

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceColorWriteEnableFeaturesEXT
• Extending VkPipelineColorBlendStateCreateInfo:
  ◦ VkPipelineColorWriteCreateInfoEXT

New Enum Constants
• VK_EXT_COLOR_WRITE_ENABLE_EXTENSION_NAME
• VK_EXT_COLOR_WRITE_ENABLE_SPEC_VERSION
• Extending VkDynamicState:
  ◦ VK_DYNAMIC_STATE_COLOR_WRITE_ENABLE_EXT
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COLOR_WRITE_ENABLE_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PIPELINE_COLOR_WRITE_CREATE_INFO_EXT
Version History

- Revision 1, 2020-01-25 (Sharif Elcott)
  - Internal revisions

VK_EXT_conditional_rendering

Name String
VK_EXT_conditional_rendering

Extension Type
Device extension

Registered Extension Number
82

Revision
2

Extension and Version Dependencies
- Requires Vulkan 1.0

Contact
- Vikram Kushwaha vkushwaha

Other Extension Metadata

Last Modified Date
2018-05-21

IP Status
No known IP claims.

Contributors
- Vikram Kushwaha, NVIDIA
- Daniel Rakos, AMD
- Jesse Hall, Google
- Jeff Bolz, NVIDIA
- Piers Daniell, NVIDIA
- Stuart Smith, Imagination Technologies

Description

This extension allows the execution of one or more rendering commands to be conditional on a value in buffer memory. This may help an application reduce the latency by conditionally discarding rendering commands without application intervention. The conditional rendering
commands are limited to draws, compute dispatches and clearing attachments within a conditional rendering block.

**New Commands**

- `vkCmdBeginConditionalRenderingEXT`
- `vkCmdEndConditionalRenderingEXT`

**New Structures**

- `VkConditionalRenderingBeginInfoEXT`
- Extending `VkCommandBufferInheritanceInfo`:
  - `VkCommandBufferInheritanceConditionalRenderingInfoEXT`
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceConditionalRenderingFeaturesEXT`

**New Enums**

- `VkConditionalRenderingFlagBitsEXT`

**New Bitmasks**

- `VkConditionalRenderingFlagsEXT`

**New Enum Constants**

- `VK_EXT_CONDITIONAL_RENDERING_EXTENSION_NAME`
- `VK_EXT_CONDITIONAL_RENDERING_SPEC_VERSION`
- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_CONDITIONAL_RENDERING_READ_BIT_EXT`
- Extending `VkBufferUsageFlagBits`:
  - `VK_BUFFER_USAGE_CONDITIONAL_RENDERING_BIT_EXT`
- Extending `VkPipelineStageFlagBits`:
  - `VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_CONDITIONAL_RENDERING_INFO_EXT`
  - `VK_STRUCTURE_TYPE_CONDITIONAL_RENDERING_BEGIN_INFO_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONDITIONAL_RENDERING_FEATURES_EXT`

**Issues**

1) Should conditional rendering affect copy and blit commands?
RESOLVED: Conditional rendering should not affect copies and blits.

2) Should secondary command buffers be allowed to execute while conditional rendering is active in the primary command buffer?

RESOLVED: The rendering commands in secondary command buffer will be affected by an active conditional rendering in primary command buffer if the conditionalRenderingEnable is set to VK_TRUE. Conditional rendering must not be active in the primary command buffer if conditionalRenderingEnable is VK_FALSE.

Examples

None.

Version History

- Revision 1, 2018-04-19 (Vikram Kushwaha)
  - First Version
- Revision 2, 2018-05-21 (Vikram Kushwaha)
  - Add new pipeline stage, access flags and limit conditional rendering to a subpass or entire renderpass.

VK_EXT_conservative_rasterization

Name String

VK_EXT_conservative_rasterization

Extension Type

Device extension

Registered Extension Number

102

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Piers Daniell @pdaniell-nv

Other Extension Metadata

Last Modified Date

2020-06-09
Interactions and External Dependencies

- This extension requires `SPV_EXT_fragment_fully_covered` if the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::fullyCoveredFragmentShaderInputVariable` feature is used.
- This extension requires `SPV_KHR_post_depth_coverage` if the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::conservativeRasterizationPostDepthCoverage` feature is used.
- This extension provides API support for `GL_NV_conservative_raster_underestimation` if the `VkPhysicalDeviceConservativeRasterizationPropertiesEXT::fullyCoveredFragmentShaderInputVariable` feature is used.

Contributors

- Daniel Koch, NVIDIA
- Daniel Rakos, AMD
- Jeff Bolz, NVIDIA
- Slawomir Grajewski, Intel
- Stu Smith, Imagination Technologies

Description

This extension adds a new rasterization mode called conservative rasterization. There are two modes of conservative rasterization; overestimation and underestimation.

When overestimation is enabled, if any part of the primitive, including its edges, covers any part of the rectangular pixel area, including its sides, then a fragment is generated with all coverage samples turned on. This extension allows for some variation in implementations by accounting for differences in overestimation, where the generating primitive size is increased at each of its edges by some sub-pixel amount to further increase conservative pixel coverage. Implementations can allow the application to specify an extra overestimation beyond the base overestimation the implementation already does. It also allows implementations to either cull degenerate primitives or rasterize them.

When underestimation is enabled, fragments are only generated if the rectangular pixel area is fully covered by the generating primitive. If supported by the implementation, when a pixel rectangle is fully covered the fragment shader input variable builtin called `FullyCoveredEXT` is set to true. The shader variable works in either overestimation or underestimation mode.

Implementations can process degenerate triangles and lines by either discarding them or generating conservative fragments for them. Degenerate triangles are those that end up with zero area after the rasterizer quantizes them to the fixed-point pixel grid. Degenerate lines are those with zero length after quantization.

New Structures

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceConservativeRasterizationPropertiesEXT`
Extending `VkPipelineRasterizationStateCreateInfo`:
   - `VkPipelineRasterizationConservativeStateCreateInfoEXT`

**New Enums**
- `VkConservativeRasterizationModeEXT`

**New Bitmasks**
- `VkPipelineRasterizationConservativeStateCreateFlagsEXT`

**New Enum Constants**
- `VK_EXT_CONSERVATIVE_RASTERIZATION_EXTENSION_NAME`
- `VK_EXT_CONSERVATIVE_RASTERIZATION_SPEC_VERSION`
 decades extending `VkStructureType`:
   - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CONSERVATIVE_RASTERIZATION_PROPERTIES_EXT`
   - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_CONSERVATIVE_STATE_CREATE_INFO_EXT`

**Version History**
- Revision 1.1, 2020-09-06 (Piers Daniell)
  - Add missing SPIR-V and GLSL dependencies.
- Revision 1, 2017-08-28 (Piers Daniell)
  - Internal revisions

**VK_EXT_custom_border_color**

**Name String**
- `VK_EXT_custom_border_color`

**Extension Type**
- Device extension

**Registered Extension Number**
- 288

**Revision**
- 12

**Extension and Version Dependencies**
- Requires Vulkan 1.0

**Contact**
- Liam Middlebrook 🌐 liam-middlebrook
Other Extension Metadata

Last Modified Date
2020-04-16

IP Status
No known IP claims.

Contributors
- Joshua Ashton, Valve
- Hans-Kristian Arntzen, Valve
- Philip Rebohle, Valve
- Liam Middlebrook, NVIDIA
- Jeff Bolz, NVIDIA
- Tobias Hector, AMD
- Jason Ekstrand, Intel
- Spencer Fricke, Samsung Electronics
- Graeme Leese, Broadcom
- Jesse Hall, Google
- Jan-Harald Fredriksen, ARM
- Tom Olson, ARM
- Stuart Smith, Imagination Technologies
- Donald Scorgie, Imagination Technologies
- Alex Walters, Imagination Technologies
- Peter Quayle, Imagination Technologies

Description

This extension provides cross-vendor functionality to specify a custom border color for use when the sampler address mode `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER` is used.

To create a sampler which uses a custom border color set `VkSamplerCreateInfo::borderColor` to one of:

- `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT`
- `VK_BORDER_COLOR_INT_CUSTOM_EXT`

When `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT` or `VK_BORDER_COLOR_INT_CUSTOM_EXT` is used, applications must provide a `VkSamplerCustomBorderCreateInfoEXT` in the pNext chain for `VkSamplerCreateInfo`. 
New Structures

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceCustomBorderColorFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceCustomBorderColorPropertiesEXT`
- Extending `VkSamplerCreateInfo`:
  - `VkSamplerCustomBorderColorCreateInfoEXT`

New Enum Constants

- `VK_EXT_CUSTOM_BORDER_COLOR_EXTENSION_NAME`
- `VK_EXT_CUSTOM_BORDER_COLOR_SPEC_VERSION`
- Extending `VkBorderColor`:
  - `VK_BORDER_COLOR_FLOAT_CUSTOM_EXT`
  - `VK_BORDER_COLOR_INT_CUSTOM_EXT`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CUSTOM_BORDER_COLOR_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_SAMPLER_CUSTOM_BORDER_COLOR_CREATE_INFO_EXT`

Issues

1) Should `VkClearColorValue` be used for the border color value, or should we have our own struct/union? Do we need to specify the type of the input values for the components? This is more of a concern if `VkClearColorValue` is used here because it provides a union of float, int, uint types.

**RESOLVED**: Will reuse existing `VkClearColorValue` structure in order to easily take advantage of float, int, uint borderColor types.

2) For hardware which supports a limited number of border colors what happens if that number is exceeded? Should this be handled by the driver unbeknownst to the application? In Revision 1 we had solved this issue using a new Object type, however that may have lead to additional system resource consumption which would otherwise not be required.

**RESOLVED**: Added `VkPhysicalDeviceCustomBorderColorPropertiesEXT::maxCustomBorderColorSamplers` for tracking implementation-specific limit, and Valid Usage statement handling overflow.

3) Should this be supported for immutable samplers at all, or by a feature bit? Some implementations may not be able to support custom border colors on immutable samplers — is it worthwhile enabling this to work on them for implementations that can support it, or forbidding it entirely.

**RESOLVED**: Samplers created with a custom border color are forbidden from being immutable.
This resolves concerns for implementations where the custom border color is an index to a LUT instead of being directly embedded into sampler state.

4) Should UINT and SINT (unsigned integer and signed integer) border color types be separated or should they be combined into one generic INT (integer) type?

**RESOLVED:** Separating these doesn’t make much sense as the existing fixed border color types don’t have this distinction, and there is no reason in hardware to do so. This separation would also create unnecessary work and considerations for the application.

**Version History**

- Revision 1, 2019-10-10 (Joshua Ashton)
  - Internal revisions.
- Revision 2, 2019-10-11 (Liam Middlebrook)
  - Remove VkCustomBorderColor object and associated functions
  - Add issues concerning HW limitations for custom border color count
- Revision 3, 2019-10-12 (Joshua Ashton)
  - Re-expose the limits for the maximum number of unique border colors
  - Add extra details about border color tracking
  - Fix typos
- Revision 4, 2019-10-12 (Joshua Ashton)
  - Changed maxUniqueCustomBorderColors to a uint32_t from a VkDeviceSize
- Revision 5, 2019-10-14 (Liam Middlebrook)
  - Added features bit
- Revision 6, 2019-10-15 (Joshua Ashton)
  - Type-ize VK_BORDER_COLOR_CUSTOM
  - Fix const-ness on pNext of VkSamplerCustomBorderColorCreateInfoEXT
- Revision 7, 2019-11-26 (Liam Middlebrook)
  - Renamed maxUniqueCustomBorderColors to maxCustomBorderColors
- Revision 8, 2019-11-29 (Joshua Ashton)
  - Renamed borderColor member of VkSamplerCustomBorderColorCreateInfoEXT to customBorderColor
- Revision 9, 2020-02-19 (Joshua Ashton)
  - Renamed maxCustomBorderColors to maxCustomBorderColorSamplers
- Revision 10, 2020-02-21 (Joshua Ashton)
  - Added format to VkSamplerCustomBorderColorCreateInfoEXT and feature bit
- Revision 11, 2020-04-07 (Joshua Ashton)
  - Dropped UINT/SINT border color differences, consolidated types
• Revision 12, 2020-04-16 (Joshua Ashton)
  ◦ Renamed VK_BORDER_COLOR_CUSTOM_FLOAT_EXT to VK_BORDER_COLOR_FLOAT_CUSTOM_EXT for consistency

**VK_EXT_debug_utils**

**Name String**

VK_EXT_debug_utils

**Extension Type**

Instance extension

**Registered Extension Number**

129

**Revision**

2

**Extension and Version Dependencies**

• Requires Vulkan 1.0

**Special Use**

• Debugging tools

**Contact**

• Mark Young marky-lunarg

**Other Extension Metadata**

**Last Modified Date**

2020-04-03

**Revision**

2

**IP Status**

No known IP claims.

**Dependencies**

• This extension is written against version 1.0 of the Vulkan API.
  • Requires VkObjectType

**Contributors**

• Mark Young, LunarG
  • Baldur Karlsson
  • Ian Elliott, Google
Description

Due to the nature of the Vulkan interface, there is very little error information available to the developer and application. By using the `VK_EXT_debug_utils` extension, developers can obtain more information. When combined with validation layers, even more detailed feedback on the application's use of Vulkan will be provided.

This extension provides the following capabilities:

- The ability to create a debug messenger which will pass along debug messages to an application supplied callback.
- The ability to identify specific Vulkan objects using a name or tag to improve tracking.
- The ability to identify specific sections within a `VkQueue` or `VkCommandBuffer` using labels to aid organization and offline analysis in external tools.

The main difference between this extension and `VK_EXT_debug_report` and `VK_EXT_debug_marker` is that those extensions use `VkDebugReportObjectTypeEXT` to identify objects. This extension uses the core `VkObjectType` in place of `VkDebugReportObjectTypeEXT`. The primary reason for this move is that no future object type handle enumeration values will be added to `VkDebugReportObjectTypeEXT` since the creation of `VkObjectType`. In addition, this extension combines the functionality of both `VK_EXT_debug_report` and `VK_EXT_debug_marker` by allowing object name and debug markers (now called labels) to be returned to the application's callback function. This should assist in clarifying the details of a debug message including: what objects are involved and potentially which location within a `VkQueue` or `VkCommandBuffer` the message occurred.

New Object Types

- `VkDebugUtilsMessengerEXT`
New Commands

- `vkCmdBeginDebugUtilsLabelEXT`
- `vkCmdEndDebugUtilsLabelEXT`
- `vkCmdInsertDebugUtilsLabelEXT`
- `vkCreateDebugUtilsMessengerEXT`
- `vkDestroyDebugUtilsMessengerEXT`
- `vkQueueBeginDebugUtilsLabelEXT`
- `vkQueueEndDebugUtilsLabelEXT`
- `vkQueueInsertDebugUtilsLabelEXT`
- `vkSetDebugUtilsObjectNameEXT`
- `vkSetDebugUtilsObjectTagEXT`
- `vkSubmitDebugUtilsMessageEXT`

New Structures

- `VkDebugUtilsLabelEXT`
- `VkDebugUtilsMessengerCallbackDataEXT`
- `VkDebugUtilsObjectNameInfoEXT`
- `VkDebugUtilsObjectTagInfoEXT`
- Extending `VkInstanceCreateInfo`:
  - `VkDebugUtilsMessengerCreateInfoEXT`

New Function Pointers

- `PFN_vkDebugUtilsMessengerCallbackEXT`

New Enums

- `VkDebugUtilsMessageSeverityFlagBitsEXT`
- `VkDebugUtilsMessageTypeFlagBitsEXT`

New Bitmasks

- `VkDebugUtilsMessageSeverityFlagsEXT`
- `VkDebugUtilsMessageTypeFlagsEXT`
- `VkDebugUtilsMessengerCallbackDataFlagsEXT`
- `VkDebugUtilsMessengerCreateFlagsEXT`
New Enum Constants

- VK_EXT_DEBUG_UTILS_EXTENSION_NAME
- VK_EXT_DEBUG_UTILS_SPEC_VERSION

Extending VkObjectType:
- VK_OBJECT_TYPE_DEBUG_UTILS_MESSENGER_EXT

Extending VkStructureType:
- VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT
- VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CALLBACK_DATA_EXT
- VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT
- VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT
- VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_TAG_INFO_EXT

Examples

Example 1

VK_EXT_debug_utils allows an application to register multiple callbacks with any Vulkan component wishing to report debug information. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. An application can register callbacks even when no validation layers are enabled, but they will only be called for loader and, if implemented, driver events.

To capture events that occur while creating or destroying an instance an application can link a VkDebugUtilsMessengerCreateInfoEXT structure to the pNext element of the VkInstanceCreateInfo structure given to vkCreateInstance. This callback is only valid for the duration of the vkCreateInstance and the vkDestroyInstance call. Use vkCreateDebugUtilsMessengerEXT to create persistent callback objects.

Example uses: Create three callback objects. One will log errors and warnings to the debug console using Windows OutputDebugString. The second will cause the debugger to break at that callback when an error happens and the third will log warnings to stdout.

```c
extern VkInstance instance;
VkResult res;
VkDebugUtilsMessengerEXT cb1, cb2, cb3;

// Must call extension functions through a function pointer:
PFN_vkCreateDebugUtilsMessengerEXT pfnCreateDebugUtilsMessengerEXT = (PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
PFN_vkDestroyDebugUtilsMessengerEXT pfnDestroyDebugUtilsMessengerEXT = (PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT");

VkDebugUtilsMessengerCreateInfoEXT callback1 = {
```
VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT,  // sType
NULL,  // pNext
0,  // flags
VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT |  // messageSeverity
VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT,
VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |  // messageType
VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT,
myOutputDebugString,  // pfnUserCallback
NULL  // pUserData
};

res = pfnCreateDebugUtilsMessengerEXT(instance, &callback1, NULL, &cb1);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

callback1.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
callback1.pfnUserCallback = myDebugBreak;
callback1.pUserData = NULL;
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback1, NULL, &cb2);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

VkDebugUtilsMessengerCreateInfoEXT callback3 = {
    VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT,  // sType
    NULL,  // pNext
    0,  // flags
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT,  // messageSeverity
    VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |  // messageType
    VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT,
    mystdOutLogger,  // pfnUserCallback
    NULL  // pUserData
};
res = pfnCreateDebugUtilsMessengerEXT(instance, &callback3, NULL, &cb3);
if (res != VK_SUCCESS) {
    // Do error handling for VK_ERROR_OUT_OF_MEMORY
}

// Remove callbacks when cleaning up
pfnDestroyDebugUtilsMessengerEXT(instance, cb1, NULL);
pfnDestroyDebugUtilsMessengerEXT(instance, cb2, NULL);
pfnDestroyDebugUtilsMessengerEXT(instance, cb3, NULL);
Associate a name with an image, for easier debugging in external tools or with validation layers that can print a friendly name when referring to objects in error messages.

```c
extern VkInstance instance;
extern VkDevice device;
extern VkImage image;

// Must call extension functions through a function pointer:
PFN_vkSetDebugUtilsObjectNameEXT pfnSetDebugUtilsObjectNameEXT =
    (PFN_vkSetDebugUtilsObjectNameEXT)vkGetInstanceProcAddr(instance,
               "vkSetDebugUtilsObjectNameEXT");

// Set a name on the image
const VkDebugUtilsObjectNameInfoEXT imageNameInfo =
    {
        VK_STRUCTURE_TYPE_DEBUG_UTILS_OBJECT_NAME_INFO_EXT, // sType
        NULL, // pNext
        VK_OBJECT_TYPE_IMAGE, // objectType
        (uint64_t)image, // object
        "Brick Diffuse Texture", // pObjectName
    };

pfnSetDebugUtilsObjectNameEXT(device, &imageNameInfo);
```

// A subsequent error might print:
// Image 'Brick Diffuse Texture' (0xc0dec0dedeadbeef) is used in a
// command buffer with no memory bound to it.

**Example 3**

Annotating regions of a workload with naming information so that offline analysis tools can display a more usable visualization of the commands submitted.

```c
extern VkInstance instance;
extern VkCommandBuffer commandBuffer;

// Must call extension functions through a function pointer:
PFN_vkQueueBeginDebugUtilsLabelEXT pfnQueueBeginDebugUtilsLabelEXT =
    (PFN_vkQueueBeginDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
               "vkQueueBeginDebugUtilsLabelEXT");
PFN_vkQueueEndDebugUtilsLabelEXT pfnQueueEndDebugUtilsLabelEXT =
    (PFN_vkQueueEndDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
               "vkQueueEndDebugUtilsLabelEXT");
PFN_vkCmdBeginDebugUtilsLabelEXT pfnCmdBeginDebugUtilsLabelEXT =
    (PFN_vkCmdBeginDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
               "vkCmdBeginDebugUtilsLabelEXT");
PFN_vkCmdEndDebugUtilsLabelEXT pfnCmdEndDebugUtilsLabelEXT =
    (PFN_vkCmdEndDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance,
               "vkCmdEndDebugUtilsLabelEXT");
```
PFN_vkCmdInsertDebugUtilsLabelEXT pfnCmdInsertDebugUtilsLabelEXT = (PFN_vkCmdInsertDebugUtilsLabelEXT)vkGetInstanceProcAddr(instance, "vkCmdInsertDebugUtilsLabelEXT");

// Describe the area being rendered
const VkDebugUtilsLabelEXT houseLabel =
{
    VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
    NULL,                                     // pNext
    "Brick House",                           // pLabelName
    { 1.0f, 0.0f, 0.0f, 1.0f },              // color
};

// Start an annotated group of calls under the 'Brick House' name
pfnCmdBeginDebugUtilsLabelEXT(commandBuffer, &houseLabel);
{
    // A mutable structure for each part being rendered
    VkDebugUtilsLabelEXT housePartLabel =
    {
        VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
        NULL,                                     // pNext
        NULL,                                     // pLabelName
        { 0.0f, 0.0f, 0.0f, 0.0f },              // color
    };

    // Set the name and insert the marker
    housePartLabel.pLabelName = "Walls";
    pfnCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

    // Insert the drawcall for the walls
    vkCmdDrawIndexed(commandBuffer, 1000, 1, 0, 0, 0);

    // Insert a recursive region for two sets of windows
    housePartLabel.pLabelName = "Windows";
    pfnCmdBeginDebugUtilsLabelEXT(commandBuffer, &housePartLabel);
    {
        vkCmdDrawIndexed(commandBuffer, 75, 6, 1000, 0, 0);
        vkCmdDrawIndexed(commandBuffer, 100, 2, 1450, 0, 0);
    }
    pfnCmdEndDebugUtilsLabelEXT(commandBuffer);

    housePartLabel.pLabelName = "Front Door";
    pfnCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

    vkCmdDrawIndexed(commandBuffer, 350, 1, 1650, 0, 0);

    housePartLabel.pLabelName = "Roof";
    pfnCmdInsertDebugUtilsLabelEXT(commandBuffer, &housePartLabel);

    vkCmdDrawIndexed(commandBuffer, 500, 1, 2000, 0, 0);
}

// End the house annotation started above
pfnCmdEndDebugUtilsLabelEXT(commandBuffer);

// Do other work
vkEndCommandBuffer(commandBuffer);

// Describe the queue being used
const VkDebugUtilsLabelEXT queueLabel = {
    VK_STRUCTURE_TYPE_DEBUG_UTILS_LABEL_EXT, // sType
    NULL, // pNext
    "Main Render Work", // pLabelName
    {0.0f, 1.0f, 0.0f, 1.0f}, // color
};

// Identify the queue label region
pfnQueueBeginDebugUtilsLabelEXT(queue, &queueLabel);

// Submit the work for the main render thread
const VkCommandBuffer cmd_bufs[] = {commandBuffer};
VkSubmitInfo submit_info = {.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO,
    .pNext = NULL,
    .waitSemaphoreCount = 0,
    .pWaitSemaphores = NULL,
    .pWaitDstStageMask = NULL,
    .commandBufferCount = 1,
    .pCommandBuffers = cmd_bufs,
    .signalSemaphoreCount = 0,
    .pSignalSemaphores = NULL};
vkQueueSubmit(queue, 1, &submit_info, fence);

// End the queue label region
pfnQueueEndDebugUtilsLabelEXT(queue);

Issues

1) Should we just name this extension VK_EXT_debug_report2

RESOLVED: No. There is enough additional changes to the structures to break backwards compatibility. So, a new name was decided that would not indicate any interaction with the previous extension.

2) Will validation layers immediately support all the new features.

RESOLVED: Not immediately. As one can imagine, there is a lot of work involved with converting the validation layer logging over to the new functionality. Basic logging, as seen in the origin VK_EXT_debug_report extension will be made available immediately. However, adding the labels and object names will take time. Since the priority for Khronos at this time is to continue focusing on Valid Usage statements, it may take a while before the new functionality is fully exposed.
3) If the validation layers won't expose the new functionality immediately, then what's the point of this extension?

**RESOLVED:** We needed a replacement for VK_EXT_debug_report because the VkDebugReportObjectTypeEXT enumeration will no longer be updated and any new objects will need to be debugged using the new functionality provided by this extension.

4) Should this extension be split into two separate parts (1 extension that is an instance extension providing the callback functionality, and another device extension providing the general debug marker and annotation functionality)?

**RESOLVED:** No, the functionality for this extension is too closely related. If we did split up the extension, where would the structures and enums live, and how would you define that the device behavior in the instance extension is really only valid if the device extension is enabled, and the functionality is passed in. It's cleaner to just define this all as an instance extension, plus it allows the application to enable all debug functionality provided with one enable string during vkCreateInstance.

**Version History**

- Revision 1, 2017-09-14 (Mark Young and all listed Contributors)
  - Initial draft, based on VK_EXT_debug_report and VK_EXT_debug_marker in addition to previous feedback supplied from various companies including Valve, Epic, and Oxide games.
- Revision 2, 2020-04-03 (Mark Young and Piers Daniell)
  - Updated to allow either *NULL* or an empty string to be passed in for pObjectName in VkDebugUtilsObjectNameInfoEXT, because the loader and various drivers support *NULL* already.

**VK_EXT_depth_clip_enable**

**Name String**

- VK_EXT_depth_clip_enable

**Extension Type**

- Device extension

**Registered Extension Number**

- 103

**Revision**

- 1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Special Use**

- *D3D support*
Description

This extension allows the depth clipping operation, that is normally implicitly controlled by `VkPipelineRasterizationStateCreateInfo::depthClampEnable`, to instead be controlled explicitly by `VkPipelineRasterizationDepthClipStateCreateInfoEXT::depthClipEnable`.

This is useful for translating DX content which assumes depth clamping is always enabled, but depth clip can be controlled by the DepthClipEnable rasterization state (D3D12_RASTERIZER_DESC).

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceDepthClipEnableFeaturesEXT`
- Extending `VkPipelineRasterizationStateCreateInfo`:
  - `VkPipelineRasterizationDepthClipStateCreateInfoEXT`

New Bitmasks

- `VkPipelineRasterizationDepthClipStateCreateFlagsEXT`

New Enum Constants

- `VK_EXT_DEPTH_CLIP_ENABLE_EXTENSION_NAME`
- `VK_EXT_DEPTH_CLIP_ENABLE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEPTH_CLIP_ENABLE_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_DEPTH_CLIP_STATE_CREATE_INFO_EXT`
**Version History**

- Revision 1, 2018-12-20 (Piers Daniell)
  - Internal revisions

**VK_EXT_depth_range_unrestricted**

**Name String**

VK_EXT_depth_range_unrestricted

**Extension Type**

Device extension

**Registered Extension Number**

14

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Piers Daniell [pdaniell-nv](mailto:pdaniell-nv)

**Other Extension Metadata**

**Last Modified Date**

2017-06-22

**Contributors**

- Daniel Koch, NVIDIA
- Jeff Bolz, NVIDIA

**Description**

This extension removes the `VkViewport minDepth` and `maxDepth` restrictions that the values must be between 0.0 and 1.0, inclusive. It also removes the same restriction on `VkPipelineDepthStencilStateCreateInfo minDepthBounds` and `maxDepthBounds`. Finally it removes the restriction on the `depth` value in `VkClearDepthStencilValue`.

**New Enum Constants**

- `VK_EXT_DEPTH_RANGE_UNRESTRICTED_EXTENSION_NAME`
- `VK_EXT_DEPTH_RANGE_UNRESTRICTED_SPEC_VERSION`
Issues

1) How do VkViewport minDepth and maxDepth values outside of the 0.0 to 1.0 range interact with Primitive Clipping?

RESOLVED: The behavior described in Primitive Clipping still applies. If depth clamping is disabled the depth values are still clipped to \(0 \leq z_c \leq w_c\) before the viewport transform. If depth clamping is enabled the above equation is ignored and the depth values are instead clamped to the VkViewport minDepth and maxDepth values, which in the case of this extension can be outside of the 0.0 to 1.0 range.

2) What happens if a resulting depth fragment is outside of the 0.0 to 1.0 range and the depth buffer is fixed-point rather than floating-point?

RESOLVED: The supported range of a fixed-point depth buffer is 0.0 to 1.0 and depth fragments are clamped to this range.

Version History

- Revision 1, 2017-06-22 (Piers Daniell)
  - Internal revisions

VK_EXT_descriptor_indexing

Name String

VK_EXT_descriptor_indexing

Extension Type

Device extension

Registered Extension Number

162

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2
- Requires VK_KHR_maintenance3

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Jeff Bolz jeffbolznv
Other Extension Metadata

Last Modified Date
2017-10-02

Interactions and External Dependencies

• Promoted to Vulkan 1.2 Core

Contributors

• Jeff Bolz, NVIDIA
• Daniel Rakos, AMD
• Slawomir Grajewski, Intel
• Tobias Hector, Imagination Technologies

Description

This extension adds several small features which together enable applications to create large descriptor sets containing substantially all of their resources, and selecting amongst those resources with dynamic (non-uniform) indexes in the shader. There are feature enables and SPIR-V capabilities for non-uniform descriptor indexing in the shader, and non-uniform indexing in the shader requires use of a new `NonUniformEXT` decoration defined in the `SPV_EXT_descriptor_indexing` SPIR-V extension. There are descriptor set layout binding creation flags enabling several features:

• Descriptors can be updated after they are bound to a command buffer, such that the execution of the command buffer reflects the most recent update to the descriptors.

• Descriptors that are not used by any pending command buffers can be updated, which enables writing new descriptors for frame N+1 while frame N is executing.

• Relax the requirement that all descriptors in a binding that is “statically used” must be valid, such that descriptors that are not accessed by a submission need not be valid and can be updated while that submission is executing.

• The final binding in a descriptor set layout can have a variable size (and unsized arrays of resources are allowed in the `GL_EXT_nonuniform_qualifier` and `SPV_EXT_descriptor_indexing` extensions).

Note that it is valid for multiple descriptor arrays in a shader to use the same set and binding number, as long as they are all compatible with the descriptor type in the pipeline layout. This means a single array binding in the descriptor set can serve multiple texture dimensionalities, or an array of buffer descriptors can be used with multiple different block layouts.

There are new descriptor set layout and descriptor pool creation flags that are required to opt in to the update-after-bind functionality, and there are separate `maxPerStage` and `maxDescriptorSet` limits that apply to these descriptor set layouts which `may` be much higher than the pre-existing limits. The old limits only count descriptors in non-updateAfterBind descriptor set layouts, and the new limits count descriptors in all descriptor set layouts in the pipeline layout.
New Structures

• Extending VkDescriptorSetAllocateInfo:
  ◦ VkDescriptorSetVariableDescriptorCountAllocateInfoEXT

• Extending VkDescriptorSetLayoutCreateInfo:
  ◦ VkDescriptorSetLayoutBindingFlagsCreateInfoEXT

• Extending VkDescriptorSetLayoutSupport:
  ◦ VkDescriptorSetVariableDescriptorCountLayoutSupportEXT

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceDescriptorIndexingFeaturesEXT

• Extending VkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceDescriptorIndexingPropertiesEXT

New Enums

• VkDescriptorBindingFlagBitsEXT

New Bitmasks

• VkDescriptorBindingFlagsEXT

New Enum Constants

• VK_EXT_DESCRIPTOR_INDEXING_EXTENSION_NAME
• VK_EXT_DESCRIPTOR_INDEXING_SPEC_VERSION

• Extending VkDescriptorBindingFlagBits:
  ◦ VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT_EXT
  ◦ VK_DESCRIPTOR_BINDING_UPDATE_AFTER_BIND_BIT_EXT
  ◦ VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT_EXT
  ◦ VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT_EXT

• Extending VkDescriptorPoolCreateFlagBits:
  ◦ VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT_EXT

• Extending VkDescriptorSetLayoutCreateFlagBits:
  ◦ VK_DESCRIPTOR_SET_LAYOUT_CREATE_UPDATE_AFTER_BIND_POOL_BIT_EXT

• Extending VkResult:
  ◦ VK_ERROR_FRAGMENTATION_EXT

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_BINDING_FLAGS_CREATE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_DESCRIPTOR_SET_VARIABLE_DESCRIPTOR_COUNT_ALLOCATE_INFO_EXT
Promotion to Vulkan 1.2

Functionality in this extension is included in core Vulkan 1.2, with the EXT suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the descriptorIndexing capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

Version History

- Revision 1, 2017-07-26 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-10-02 (Jeff Bolz)
  - ???

VK_EXT_device_memory_report

Name String

VK_EXT_device_memory_report

Extension Type

Device extension

Registered Extension Number

285

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Special Use

- Developer tools

Contact

- Yiwei Zhang zhangyiwei

Other Extension Metadata

Last Modified Date

2021-01-06
IP Status
No known IP claims.

Contributors
- Yiwei Zhang, Google
- Jesse Hall, Google

Description
This device extension allows registration of device memory event callbacks upon device creation, so that applications or middleware can obtain detailed information about memory usage and how memory is associated with Vulkan objects. This extension exposes the actual underlying device memory usage, including allocations that are not normally visible to the application, such as memory consumed by `vkCreateGraphicsPipelines`. It is intended primarily for use by debug tooling rather than for production applications.

New Structures
- `VkDeviceMemoryReportCallbackDataEXT`
- Extending `VkDeviceCreateInfo`:
  - `VkDeviceDeviceMemoryReportCreateInfoEXT`
- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceDeviceMemoryReportFeaturesEXT`

New Function Pointers
- `PFN_vkDeviceMemoryReportCallbackEXT`

New Enums
- `VkDeviceMemoryReportEventTypeEXT`

New Bitmasks
- `VkDeviceMemoryReportFlagsEXT`

New Enum Constants
- `VK_EXT_DEVICE_MEMORY_REPORT_EXTENSION_NAME`
- `VK_EXT_DEVICE_MEMORY_REPORT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DEVICE_DEVICE_MEMORY_REPORT_CREATE_INFO_EXT`
  - `VK_STRUCTURE_TYPE_DEVICE_MEMORY_REPORT_CALLBACK_DATA_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_MEMORY_REPORT_FEATURES_EXT`
Issues

1) Should this be better expressed as an extension to VK_EXT_debug_utils and its general purpose messenger construct?

**RESOLVED**: No. The intended lifecycle is quite different. We want to make this extension tied to the device’s lifecycle. Each ICD just handles its own implementation of this extension, and this extension will only be directly exposed from the ICD. So we can avoid the extra implementation complexity used to accommodate the flexibility of VK_EXT_debug_utils extension.

2) Can we extend and use the existing internal allocation callbacks instead of adding the new callback structure in this extension?

**RESOLVED**: No. Our memory reporting layer that combines this information with other memory info it collects directly (e.g. bindings of resources to VkDeviceMemory) would have to intercept all entry points that take a VkAllocationCallbacks parameter and inject its own pfnInternalAllocation and pfnInternalFree. That's maybe doable for the extensions we know about, but not for ones we don't. The proposal would work fine in the face of most unknown extensions. But even for ones we know about, since apps can provide a different set of callbacks and userdata and those can be retained by the driver and used later (esp. for pool object, but not just those), we'd have to dynamically allocate the interception trampoline every time. That's getting to be an unreasonably large amount of complexity and (possibly) overhead.

We're interested in both alloc/free and import/unimport. The latter is fairly important for tracking (and avoiding double-counting) of swapchain images (still true with “native swapchains” based on external memory) and media/camera interop. Though we might be able to handle this with additional VkInternalAllocationType values, for import/export we do want to be able to tie this to the external resource, which is one thing that the memoryObjectId is for.

The internal alloc/free callbacks are not extensible except via new VkInternalAllocationType values. The VkDeviceMemoryReportCallbackDataEXT in this extension is extensible. That was deliberate: there's a real possibility we'll want to get extra information in the future. As one example, currently this reports only physical allocations, but we believe there are interesting cases for tracking how populated that VA region is.

The callbacks are clearly specified as only callable within the context of a call from the app into Vulkan. We believe there are some cases where drivers can allocate device memory asynchronously. This was one of the sticky issues that derailed the internal device memory allocation reporting design (which is essentially what this extension is trying to do) leading up to 1.0.

**VkAllocationCallbacks** is described in a section called “Host memory” and the intro to it is very explicitly about host memory. The other callbacks are all inherently about host memory. But this extension is very focused on device memory.

3) Should the callback be reporting which heap is used?

**RESOLVED**: Yes. It's important for non-UMA systems to have all the device memory allocations attributed to the corresponding device memory heaps. For internally-allocated device memory, heapIndex will always correspond to an advertised heap, rather than having a magic value.
indicating a non-advertised heap. Drivers can advertise heaps that don’t have any corresponding memory types if they need to.

4) Should we use an array of callback for the layers to intercept instead of chaining multiple of the VkDeviceDeviceMemoryReportCreateInfoEXT structures in the pNext of VkDeviceCreateInfo?

**RESOLVED** No. The pointer to the VkDeviceDeviceMemoryReportCreateInfoEXT structure itself is const and you can’t just cast it away. Thus we can’t update the callback array inside the structure. In addition, we can’t drop this pNext chain either, so making a copy of this whole structure doesn’t work either.

5) Should we track bulk allocations shared among multiple objects?

**RESOLVED** No. Take the shader heap as an example. Some implementations will let multiple VkPipeline objects share the same shader heap. We are not asking the implementation to report VK_OBJECT_TYPE_PIPELINE along with a VK_NULL_HANDLE for this bulk allocation. Instead, this bulk allocation is considered as a layer below what this extension is interested in. Later, when the actual VkPipeline objects are created by suballocating from the bulk allocation, we ask the implementation to report the valid handles of the VkPipeline objects along with the actual suballocated sizes and different memoryObjectId.

6) Can we require the callbacks to be always called in the same thread with the Vulkan commands?

**RESOLVED** No. Some implementations might choose to multiplex work from multiple application threads into a single backend thread and perform JIT allocations as a part of that flow. Since this behavior is theoretically legit, we can’t require the callbacks to be always called in the same thread with the Vulkan commands, and the note is to remind the applications to handle this case properly.

7) Should we add an additional “allocation failed” event type with things like size and heap index reported?

**RESOLVED** Yes. This fits in well with the callback infrastructure added in this extension, and implementation touches the same code and has the same overheads as the rest of the extension. It could help debugging things like getting an VK_ERROR_OUT_OF_HOST_MEMORY error when ending a command buffer. Right now the allocation failure could have happened anywhere during recording, and a callback would be really useful to understand where and why.

**Version History**

- Revision 1, 2020-08-26 (Yiwei Zhang)
  - Initial version
- Revision 2, 2021-01-06 (Yiwei Zhang)
  - Minor description update

**VK_EXT_direct_mode_display**

**Name String**

VK_EXT_direct_mode_display
Extension Type
Instance extension

Registered Extension Number
89

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_display

Contact
• James Jones @cubanismo

Other Extension Metadata

Last Modified Date
2016-12-13

IP Status
No known IP claims.

Contributors
• Pierre Boudier, NVIDIA
• James Jones, NVIDIA
• Damien Leone, NVIDIA
• Pierre-Loup Griffais, Valve
• Liam Middlebrook, NVIDIA

Description
This is extension, along with related platform extensions, allows applications to take exclusive
control of displays associated with a native windowing system. This is especially useful for virtual
reality applications that wish to hide HMDs (head mounted displays) from the native platform’s
display management system, desktop, and/or other applications.

New Commands
• vkReleaseDisplayEXT

New Enum Constants
• VK_EXT_DIRECT_MODE_DISPLAY_EXTENSION_NAME
• VK_EXT_DIRECT_MODE_DISPLAY_SPEC_VERSION
Issues

1) Should this extension and its related platform-specific extensions leverage `VK_KHR_display`, or provide separate equivalent interfaces.

**RESOLVED**: Use `VK_KHR_display` concepts and objects. `VK_KHR_display` can be used to enumerate all displays on the system, including those attached to/in use by a window system or native platform, but `VK_KHR_display_swapchain` will fail to create a swapchain on in-use displays. This extension and its platform-specific children will allow applications to grab in-use displays away from window systems and/or native platforms, allowing them to be used with `VK_KHR_display_swapchain`.

2) Are separate calls needed to acquire displays and enable direct mode?

**RESOLVED**: No, these operations happen in one combined command. Acquiring a display puts it into direct mode.

Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft

`VK_EXT_directfb_surface`

Name String

`VK_EXT_directfb_surface`

Extension Type

Instance extension

Registered Extension Number

347

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_surface`

Contact

- Nicolas Caramelli [caramelli](https://github.com/caramelli)

Other Extension Metadata

Last Modified Date

2020-06-16
Description

The `VK_EXT_directfb_surface` extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the `VK_KHR_surface` extension) that refers to a DirectFB `IDirectFBSurface`, as well as a query to determine support for rendering via DirectFB.

New Commands

- `vkCreateDirectFBSurfaceEXT`
- `vkGetPhysicalDeviceDirectFBPresentationSupportEXT`

New Structures

- `VkDirectFBSurfaceCreateInfoEXT`

New Bitmasks

- `VkDirectFBSurfaceCreateFlagsEXT`

New Enum Constants

- `VK_EXT_DIRECTFB_SURFACE_EXTENSION_NAME`
- `VK_EXT_DIRECTFB_SURFACE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DIRECTFB_SURFACE_CREATE_INFO_EXT`

Version History

- Revision 1, 2020-06-16 (Nicolas Caramelli)
  - Initial version

`VK_EXT_discard_rectangles`

Name String

`VK_EXT_discard_rectangles`

Extension Type

Device extension
This extension provides additional orthogonally aligned “discard rectangles” specified in framebuffer-space coordinates that restrict rasterization of all points, lines and triangles.

From zero to an implementation-dependent limit (specified by maxDiscardRectangles) number of discard rectangles can be operational at once. When one or more discard rectangles are active, rasterized fragments can either survive if the fragment is within any of the operational discard rectangles (VK_DISCARD_RECTANGLE_MODE_INCLUSIVE_EXT mode) or be rejected if the fragment is within any of the operational discard rectangles (VK_DISCARD_RECTANGLE_MODE_EXCLUSIVE_EXT mode).

These discard rectangles operate orthogonally to the existing scissor test functionality. The discard rectangles can be different for each physical device in a device group by specifying the device mask and setting discard rectangle dynamic state.

New Commands

- vkCmdSetDiscardRectangleEXT
New Structures

- Extending `VkGraphicsPipelineCreateInfo`:
  - `VkPipelineDiscardRectangleStateCreateInfoEXT`

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceDiscardRectanglePropertiesEXT`

New Enums

- `VkDiscardRectangleModeEXT`

New Bitmasks

- `VkPipelineDiscardRectangleStateCreateFlagsEXT`

New Enum Constants

- `VK_EXT_DISCARD_RECTANGLES_EXTENSION_NAME`
- `VK_EXT_DISCARD_RECTANGLES_SPEC_VERSION`

- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DISCARD_RECTANGLE_PROPERTIES_EXT`
  - `VK_STRUCTURE_TYPE_PIPELINE_DISCARD_RECTANGLE_STATE_CREATE_INFO_EXT`

Version History

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

VK_EXT_display_control

Name String

`VK_EXT_display_control`

Extension Type

Device extension

Registered Extension Number

92

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_EXT_display_surface_counter
- Requires VK_KHR_swapchain

Contact

- James Jones cubanismo

Other Extension Metadata

Last Modified Date

2016-12-13

IP Status

No known IP claims.

Contributors

- Pierre Boudier, NVIDIA
- James Jones, NVIDIA
- Damien Leone, NVIDIA
- Pierre-Loup Griffais, Valve
- Daniel Vetter, Intel

Description

This extension defines a set of utility functions for use with the VK_KHR_display and VK_KHR_display_swapchain extensions.

New Commands

- vkDisplayPowerControlEXT
- vkGetSwapchainCounterEXT
- vkRegisterDeviceEventEXT
- vkRegisterDisplayEventEXT

New Structures

- VkDeviceEventInfoEXT
- VkDisplayEventInfoEXT
- VkDisplayPowerInfoEXT
- Extending VkSwapchainCreateInfoKHR:
  - VkSwapchainCounterCreateInfoEXT
New Enums

- VkDeviceEventTypeEXT
- VkDisplayEventTypeEXT
- VkDisplayPowerStateEXT

New Enum Constants

- VK_EXT_DISPLAY_CONTROL_EXTENSION_NAME
- VK_EXT_DISPLAY_CONTROL_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DEVICE_EVENT_INFO_EXT
  - VK_STRUCTURE_TYPE_DISPLAY_EVENT_INFO_EXT
  - VK_STRUCTURE_TYPE_DISPLAY_POWER_INFO_EXT
  - VK_STRUCTURE_TYPE_SWAPCHAIN_COUNTER_CREATE_INFO_EXT

Issues

1) Should this extension add an explicit “WaitForVsync” API or a fence signaled at vsync that the application can wait on?

**RESOLVED**: A fence. A separate API could later be provided that allows exporting the fence to a native object that could be inserted into standard run loops on POSIX and Windows systems.

2) Should callbacks be added for a vsync event, or in general to monitor events in Vulkan?

**RESOLVED**: No, fences should be used. Some events are generated by interrupts which are managed in the kernel. In order to use a callback provided by the application, drivers would need to have the userspace driver spawn threads that would wait on the kernel event, and hence the callbacks could be difficult for the application to synchronize with its other work given they would arrive on a foreign thread.

3) Should vblank or scanline events be exposed?

**RESOLVED**: Vblank events. Scanline events could be added by a separate extension, but the latency of processing an interrupt and waking up a userspace event is high enough that the accuracy of a scanline event would be rather low. Further, per-scanline interrupts are not supported by all hardware.

Version History

- Revision 1, 2016-12-13 (James Jones)
  - Initial draft
VK_EXT_display_surface_counter

Name String
   VK_EXT_display_surface_counter

Extension Type
   Instance extension

Registered Extension Number
   91

Revision
   1

Extension and Version Dependencies
   • Requires Vulkan 1.0
   • Requires VK_KHR_display

Contact
   • James Jones cubanismo

Other Extension Metadata

Last Modified Date
   2016-12-13

IP Status
   No known IP claims.

Contributors
   • Pierre Boudier, NVIDIA
   • James Jones, NVIDIA
   • Damien Leone, NVIDIA
   • Pierre-Loup Griffais, Valve
   • Daniel Vetter, Intel

Description
   This extension defines a vertical blanking period counter associated with display surfaces. It provides a mechanism to query support for such a counter from a VkSurfaceKHR object.

New Commands
   • vkGetPhysicalDeviceSurfaceCapabilities2EXT
New Structures

• VkSurfaceCapabilities2EXT

New Enums

• VkSurfaceCounterFlagBitsEXT

New Bitmasks

• VkSurfaceCounterFlagsEXT

New Enum Constants

• VK_EXT_DISPLAY_SURFACE_COUNTER_EXTENSION_NAME
• VK_EXT_DISPLAY_SURFACE_COUNTER_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_2_EXT

Version History

• Revision 1, 2016-12-13 (James Jones)
  ◦ Initial draft

VK_EXT_extended_dynamic_state

Name String

VK_EXT_extended_dynamic_state

Extension Type

Device extension

Registered Extension Number

268

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
  • Requires VK_KHR_get_physical_device_properties2

Contact

• Piers Daniell pdaniell-nv
Other Extension Metadata

Last Modified Date
2019-12-09

IP Status
No known IP claims.

Contributors
- Dan Ginsburg, Valve Corporation
- Graeme Leese, Broadcom
- Hans-Kristian Arntzen, Valve Corporation
- Jan-Harald Fredriksen, Arm Limited
- Jason Ekstrand, Intel
- Jeff Bolz, NVIDIA
- Jesse Hall, Google
- Philip Rebohle, Valve Corporation
- Stuart Smith, Imagination Technologies
- Tobias Hector, AMD

Description
This extension adds some more dynamic state to support applications that need to reduce the number of pipeline state objects they compile and bind.

New Commands
- vkCmdBindVertexBuffers2EXT
- vkCmdSetCullModeEXT
- vkCmdSetDepthBoundsTestEnableEXT
- vkCmdSetDepthCompareOpEXT
- vkCmdSetDepthTestEnableEXT
- vkCmdSetDepthWriteEnableEXT
- vkCmdSetFrontFaceEXT
- vkCmdSetPrimitiveTopologyEXT
- vkCmdSetScissorWithCountEXT
- vkCmdSetStencilOpEXT
- vkCmdSetStencilTestEnableEXT
- vkCmdSetViewportWithCountEXT
New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceExtendedDynamicStateFeaturesEXT`

New Enum Constants

- `VK_EXT_EXTENDED_DYNAMIC_STATE_EXTENSION_NAME`
- `VK_EXT_EXTENDED_DYNAMIC_STATE_SPEC_VERSION`

- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_CULL_MODE_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_BOUNDS_TEST_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_COMPARE_OP_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_TEST_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_DEPTH_WRITE_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_FRONT_FACE_EXT`
  - `VK_DYNAMIC_STATE_PRIMITIVE_TOPOLOGY_EXT`
  - `VK_DYNAMIC_STATE_SCISSOR_WITH_COUNT_EXT`
  - `VK_DYNAMIC_STATE_STENCIL_OP_EXT`
  - `VK_DYNAMIC_STATE_STENCIL_TEST_ENABLE_EXT`
  - `VK_DYNAMIC_STATE_VERTEX_INPUT_BINDING_STRIDE_EXT`
  - `VK_DYNAMIC_STATE_VIEWPORT_WITH_COUNT_EXT`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_FEATURES_EXT`

Version History

- Revision 1, 2019-12-09 (Piers Daniell)
  - Internal revisions

**VK_EXT_extended_dynamic_state2**

Name String

`VK_EXT_extended_dynamic_state2`

Extension Type

Device extension

Registered Extension Number

378
Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Vikram Kushwaha 🌐 vkushwaha-nv

Other Extension Metadata

Last Modified Date

2021-04-12

IP Status

No known IP claims.

Contributors

- Vikram Kushwaha, NVIDIA
- Piers Daniell, NVIDIA
- Jeff Bolz, NVIDIA

Description

This extension adds some more dynamic state to support applications that need to reduce the number of pipeline state objects they compile and bind.

New Commands

- `vkCmdSetDepthBiasEnableEXT`
- `vkCmdSetLogicOpEXT`
- `vkCmdSetPatchControlPointsEXT`
- `vkCmdSetPrimitiveRestartEnableEXT`
- `vkCmdSetRasterizerDiscardEnableEXT`

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceExtendedDynamicState2FeaturesEXT`

New Enum Constants

- `VK_EXT_EXTENDED_DYNAMIC_STATE_2_EXTENSION_NAME`
- `VK_EXT_EXTENDED_DYNAMIC_STATE_2_SPEC_VERSION`
• Extending `VkDynamicState`:
  ◦ `VK_DYNAMIC_STATE_DEPTH_BIAS_ENABLE_EXT`
  ◦ `VK_DYNAMIC_STATE_LOGIC_OP_EXT`
  ◦ `VK_DYNAMIC_STATE_PATCH_CONTROL_POINTS_EXT`
  ◦ `VK_DYNAMIC_STATE_PRIMITIVE_RESTART_ENABLE_EXT`
  ◦ `VK_DYNAMIC_STATE_RASTERIZER_DISCARD_ENABLE_EXT`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTENDED_DYNAMIC_STATE_2_FEATURES_EXT`

**Version History**

• Revision 1, 2021-04-12 (Vikram Kushwaha)
  ◦ Internal revisions

**VK_EXT_external_memory_dma_buf**

**Name String**

`VK_EXT_external_memory_dma_buf`

**Extension Type**

Device extension

**Registered Extension Number**

126

**Revision**

1

**Extension and Version Dependencies**

• Requires Vulkan 1.0
  
• Requires `VK_KHR_external_memory_fd`

**Contact**

• Chad Versace @chadversary

**Other Extension Metadata**

**Last Modified Date**

2017-10-10

**IP Status**

No known IP claims.
Contributors

• Chad Versace, Google
• James Jones, NVIDIA
• Jason Ekstrand, Intel

Description

A `dma_buf` is a type of file descriptor, defined by the Linux kernel, that allows sharing memory across kernel device drivers and across processes. This extension enables applications to import a `dma_buf` as `VkDeviceMemory`, to export `VkDeviceMemory` as a `dma_buf`, and to create `VkBuffer` objects that can be bound to that memory.

New Enum Constants

- `VK_EXT_EXTERNAL_MEMORY_DMA_BUF_EXTENSION_NAME`
- `VK_EXT_EXTERNAL_MEMORY_DMA_BUF_SPEC_VERSION`
- Extending `VkExternalMemoryHandleTypeFlagBits`:
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_DMA_BUF_BIT_EXT`

Issues

1) How does the application, when creating a `VkImage` that it intends to bind to `dma_buf` `VkDeviceMemory` containing an externally produced image, specify the memory layout (such as row pitch and DRM format modifier) of the `VkImage`? In other words, how does the application achieve behavior comparable to that provided by `EGL_EXT_image_dma_buf_import` and `EGL_EXT_image_dma_buf_import_modifiers`?

**RESOLVED:** Features comparable to those in `EGL_EXT_image_dma_buf_import` and `EGL_EXT_image_dma_buf_import_modifiers` will be provided by an extension layered atop this one.

2) Without the ability to specify the memory layout of external `dma_buf` images, how is this extension useful?

**RESOLVED:** This extension provides exactly one new feature: the ability to import/export between `dma_buf` and `VkDeviceMemory`. This feature, together with features provided by `VK_KHR_external_memory_fd`, is sufficient to bind a `VkBuffer` to `dma_buf`.

Version History

- Revision 1, 2017-10-10 (Chad Versace)
  - Squashed internal revisions

**VK_EXT_external_memory_host**

Name String

`VK_EXT_external_memory_host`
Extension Type

Device extension

Registered Extension Number

179

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_external_memory

Contact

- Daniel Rakos drakos-amd

Other Extension Metadata

Last Modified Date

2017-11-10

IP Status

No known IP claims.

Contributors

- Jaakko Konttinen, AMD
- David Mao, AMD
- Daniel Rakos, AMD
- Tobias Hector, Imagination Technologies
- Jason Ekstrand, Intel
- James Jones, NVIDIA

Description

This extension enables an application to import host allocations and host mapped foreign device memory to Vulkan memory objects.

New Commands

- vkGetMemoryHostPointerPropertiesEXT

New Structures

- VkMemoryHostPointerPropertiesEXT
- Extending VkMemoryAllocateInfo:
• VkImportMemoryHostPointerInfoEXT

• Extending VkPhysicalDeviceProperties2:
  • VkPhysicalDeviceExternalMemoryHostPropertiesEXT

New Enum Constants

• VK_EXT_EXTERNAL_MEMORY_HOST_EXTENSION_NAME
• VK_EXT_EXTERNAL_MEMORY_HOST_SPEC_VERSION

• Extending VkExternalMemoryHandleTypeFlagBits:
  • VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_ALLOCATION_BIT_EXT
  • VK_EXTERNAL_MEMORY_HANDLE_TYPE_HOST_MAPPED_FOREIGN_MEMORY_BIT_EXT

• Extending VkStructureType:
  • VKSTRUCTURETYPEIMPORTMEMORYHOSTPOINTERINFO_EXT
  • VKSTRUCTURETYPEMEMORYHOSTPOINTERPROPERTIES_EXT
  • VKSTRUCTURETYPEPHYSICALDEVICEEXTERNALMEMORYHOSTPROPERTIES_EXT

Issues

1) What memory type has to be used to import host pointers?

**RESOLVED:** Depends on the implementation. Applications have to use the new `vkGetMemoryHostPointerPropertiesEXT` command to query the supported memory types for a particular host pointer. The reported memory types may include memory types that come from a memory heap that is otherwise not usable for regular memory object allocation and thus such a heap's size may be zero.

2) Can the application still access the contents of the host allocation after importing?

**RESOLVED:** Yes. However, usual synchronization requirements apply.

3) Can the application free the host allocation?

**RESOLVED:** No, it violates valid usage conditions. Using the memory object imported from a host allocation that's already freed thus results in undefined behavior.

4) Is `vkMapMemory` expected to return the same host address which was specified when importing it to the memory object?

**RESOLVED:** No. Implementations are allowed to return the same address but it's not required. Some implementations might return a different virtual mapping of the allocation, although the same physical pages will be used.

5) Is there any limitation on the alignment of the host pointer and/or size?

**RESOLVED:** Yes. Both the address and the size have to be an integer multiple of `minImportedHostPointerAlignment`. In addition, some platforms and foreign devices may have
additional restrictions.

6) Can the same host allocation be imported multiple times into a given physical device?

**RESOLVED**: No, at least not guaranteed by this extension. Some platforms do not allow locking the same physical pages for device access multiple times, so attempting to do it may result in undefined behavior.

7) Does this extension support exporting the new handle type?

**RESOLVED**: No.

8) Should we include the possibility to import host mapped foreign device memory using this API?

**RESOLVED**: Yes, through a separate handle type. Implementations are still allowed to support only one of the handle types introduced by this extension by not returning import support for a particular handle type as returned in `VkExternalMemoryPropertiesKHR`.

**Version History**

- Revision 1, 2017-11-10 (Daniel Rakos)
  - Internal revisions

**VK_EXT_filter_cubic**

**Name String**

- `VK_EXT_filter_cubic`

**Extension Type**

- Device extension

**Registered Extension Number**

- 171

**Revision**

- 3

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Bill Licea-Kane

**Other Extension Metadata**

**Last Modified Date**

- 2019-12-13
Contributors

- Bill Licea-Kane, Qualcomm Technologies, Inc.
- Andrew Garrard, Samsung
- Daniel Koch, NVIDIA
- Donald Scorgie, Imagination Technologies
- Graeme Leese, Broadcom
- Jan-Herald Fredericksen, ARM
- Jeff Leger, Qualcomm Technologies, Inc.
- Tobias Hector, AMD
- Tom Olson, ARM
- Stuart Smith, Imagination Technologies

Description

**VK_EXT_filter_cubic** extends **VK_IMG_filter_cubic**.

It documents cubic filtering of other image view types. It adds new structures that can be added to the **pNext** chain of **VkPhysicalDeviceImageFormatInfo2** and **VkImageFormatProperties2** that can be used to determine which image types and which image view types support cubic filtering.

New Structures

- Extending **VkImageFormatProperties2**:
  - **VkFilterCubicImageViewImageFormatPropertiesEXT**
- Extending **VkPhysicalDeviceImageFormatInfo2**:
  - **VkPhysicalDeviceImageViewImageFormatInfoEXT**

New Enum Constants

- **VK_EXT_FILTER_CUBIC_EXTENSION_NAME**
- **VK_EXT_FILTER_CUBIC_SPEC_VERSION**
- Extending **VkFilter**:
  - **VK_FILTER_CUBIC_EXT**
- Extending **VkFormatFeatureFlagBits**:
  - **VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_EXT**
- Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_FILTER_CUBIC_IMAGE_VIEW_IMAGE_FORMAT_PROPERTIES_EXT**
  - **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_VIEW_IMAGE_FORMAT_INFO_EXT**
Version History

- Revision 3, 2019-12-13 (wwlk)
  - Delete requirement to cubic filter the formats USCALED_PACKED32, SSCALED_PACKED32, UINT_PACK32, and SINT_PACK32 (cut/paste error)
- Revision 2, 2019-06-05 (wwlk)
  - Clarify 1D optional
- Revision 1, 2019-01-24 (wwlk)
  - Initial version

VK_EXT_fragment_density_map

Name String

VK_EXT_fragment_density_map

Extension Type

Device extension

Registered Extension Number

219

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Matthew Netsch @mnetsch

Other Extension Metadata

Last Modified Date

2018-09-25

Interactions and External Dependencies

- This extension requires SPV_EXT_fragment_invocation_density

Contributors

- Matthew Netsch, Qualcomm Technologies, Inc.
- Robert VanReenen, Qualcomm Technologies, Inc.
- Jonathan Wicks, Qualcomm Technologies, Inc.
- Tate Hornbeck, Qualcomm Technologies, Inc.
Description

This extension allows an application to specify areas of the render target where the fragment shader may be invoked fewer times. These fragments are broadcasted out to multiple pixels to cover the render target.

The primary use of this extension is to reduce workloads in areas where lower quality may not be perceived such as the distorted edges of a lens or the periphery of a user's gaze.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceFragmentDensityMapFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceFragmentDensityMapPropertiesEXT`
- Extending `VkRenderPassCreateInfo`, `VkRenderPassCreateInfo2`:
  - `VkRenderPassFragmentDensityMapCreateInfoEXT`

New Enum Constants

- `VK_EXT_FRAGMENT_DENSITY_MAP_EXTENSION_NAME`
- `VK_EXT_FRAGMENT_DENSITY_MAP_SPEC_VERSION`
- Extending `VkAccessFlagBits`:
  - `VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT`
- Extending `VkFormatFeatureFlagBits`:
  - `VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT`
- Extending `VkImageLayout`:
  - `VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT`
- Extending `VkImageUsageFlagBits`:
  - `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`
• Extending VkImageViewCreateFlagBits:
  ◦ VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT

• Extending VkPipelineStageFlagBits:
  ◦ VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT

• Extending VkSamplerCreateFlagBits:
  ◦ VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT
  ◦ VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_RENDER_PASS_FRAGMENT_DENSITY_MAP_CREATE_INFO_EXT

New or Modified Built-In Variables

• FragInvocationCountEXT
• FragSizeEXT

New SPIR-V Capabilities

• FragmentDensityEXT

Version History

• Revision 1, 2018-09-25 (Matthew Netsch)
  ◦ Initial version

VK_EXT_fragment_density_map2

Name String
  VK_EXT_fragment_density_map2

Extension Type
  Device extension

Registered Extension Number
  333

Revision
  1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_EXT_fragment_density_map
Description

This extension adds additional features and properties to VK_EXT_fragment_density_map in order to reduce fragment density map host latency as well as improved queries for subsampled sampler implementation-dependent behavior.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceFragmentDensityMapFeaturesEXT
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceFragmentDensityMapPropertiesEXT
- Extending VkRenderPassCreateInfo, VkRenderPassCreateInfo2:
  - VkRenderPassFragmentDensityMapCreateInfoEXT

New Enum Constants

- VK_EXT_FRAGMENT_DENSITY_MAP_EXTENSION_NAME
- VK_EXT_FRAGMENT_DENSITY_MAP_SPEC_VERSION
- Extending VkAccessFlagBits:
  - VK_ACCESS_FRAGMENT_DENSITY_MAP_READ_BIT_EXT
- Extending VkFormatFeatureFlagBits:
  - VK_FORMAT_FEATURE_FRAGMENT_DENSITY_MAP_BIT_EXT
- Extending VkImageCreateFlagBits:
  - VK_IMAGE_CREATE_SUBSAMPLED_BIT_EXT
• Extending `VkImageLayout`:
  ◦ `VK_IMAGE_LAYOUT_FRAGMENT_DENSITY_MAP_OPTIMAL_EXT`

• Extending `VkImageUsageFlagBits`:
  ◦ `VK_IMAGE_USAGE_FRAGMENT_DENSITY_MAP_BIT_EXT`

• Extending `VkImageViewCreateFlagBits`:
  ◦ `VK_IMAGE_VIEW_CREATE_FRAGMENT_DENSITY_MAP_DYNAMIC_BIT_EXT`

• Extending `VkPipelineStageFlagBits`:
  ◦ `VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT`

• Extending `VkSamplerCreateFlagBits`:
  ◦ `VK_SAMPLER_CREATE_SUBSAMPLED_BIT_EXT`
  ◦ `VK_SAMPLER_CREATE_SUBSAMPLED_COARSE_RECONSTRUCTION_BIT_EXT`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_FEATURES_EXT`
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_DENSITY_MAP_PROPERTIES_EXT`
  ◦ `VK_STRUCTURE_TYPE_RENDER_PASS_FRAGMENT_DENSITY_MAP_CREATE_INFO_EXT`

**Version History**

• Revision 1, 2020-06-16 (Matthew Netsch)
  ◦ Initial version

**VK_EXT_fragment_shader_interlock**

**Name String**

`VK_EXT_fragment_shader_interlock`

**Extension Type**

Device extension

**Registered Extension Number**

252

**Revision**

1

**Extension and Version Dependencies**

• Requires Vulkan 1.0
• Requires `VK_KHR_get_physical_device_properties2`

**Contact**

• Piers Daniell [pdaniell-nv](#)
Other Extension Metadata

Last Modified Date
2019-05-02

Interactions and External Dependencies
• This extension requires SPV_EXT_fragment_shader_interlock
• This extension provides API support for GL_ARB_fragment_shader_interlock

Contributors
• Daniel Koch, NVIDIA
• Graeme Leese, Broadcom
• Jan-Harald Fredriksen, Arm
• Jason Ekstrand, Intel
• Jeff Bolz, NVIDIA
• Ruihao Zhang, Qualcomm
• Slawomir Grajewski, Intel
• Spencer Fricke, Samsung

Description
This extension adds support for the FragmentShaderPixelInterlockEXT, FragmentShaderSampleInterlockEXT, and FragmentShaderShadingRateInterlockEXT capabilities from the SPV_EXT_fragment_shader_interlock extension to Vulkan.

Enabling these capabilities provides a critical section for fragment shaders to avoid overlapping pixels being processed at the same time, and certain guarantees about the ordering of fragment shader invocations of fragments of overlapping pixels.

This extension can be useful for algorithms that need to access per-pixel data structures via shader loads and stores. Algorithms using this extension can access per-pixel data structures in critical sections without other invocations accessing the same per-pixel data. Additionally, the ordering guarantees are useful for cases where the API ordering of fragments is meaningful. For example, applications may be able to execute programmable blending operations in the fragment shader, where the destination buffer is read via image loads and the final value is written via image stores.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceFragmentShaderInterlockFeaturesEXT

New Enum Constants
• VK_EXT_FRAGMENT_SHADER_INTERLOCK_EXTENSION_NAME
• VK_EXT_FRAGMENT_SHADER_INTERLOCK_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_INTERLOCK_FEATURES_EXT

New SPIR-V Capabilities

• FragmentShaderInterlockEXT
• FragmentShaderPixelInterlockEXT
• FragmentShaderShadingRateInterlockEXT

Version History

• Revision 1, 2019-05-24 (Piers Daniell)
  ◦ Internal revisions

VK_EXT_full_screen_exclusive

Name String

VK_EXT_full_screen_exclusive

Extension Type

Device extension

Registered Extension Number

256

Revision

4

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2
• Requires VK_KHR_surface
• Requires VK_KHR_get_surface_capabilities2
• Requires VK_KHR_swapchain

Contact

• James Jones cubanismo

Other Extension Metadata

Last Modified Date

2019-03-12

IP Status

No known IP claims.
Interactions and External Dependencies

- Interacts with Vulkan 1.1
- Interacts with VK_KHR_device_group
- Interacts with VK_KHR_win32_surface

Contributors

- Hans-Kristian Arntzen, ARM
- Slawomir Grajewski, Intel
- Tobias Hector, AMD
- James Jones, NVIDIA
- Daniel Rakos, AMD
- Jeff Juliano, NVIDIA
- Joshua Schnarr, NVIDIA
- Aaron Hagan, AMD

Description

This extension allows applications to set the policy for swapchain creation and presentation mechanisms relating to full-screen access. Implementations may be able to acquire exclusive access to a particular display for an application window that covers the whole screen. This can increase performance on some systems by bypassing composition, however it can also result in disruptive or expensive transitions in the underlying windowing system when a change occurs.

Applications can choose between explicitly disallowing or allowing this behavior, letting the implementation decide, or managing this mode of operation directly using the new `vkAcquireFullScreenExclusiveModeEXT` and `vkReleaseFullScreenExclusiveModeEXT` commands.

New Commands

- `vkAcquireFullScreenExclusiveModeEXT`
- `vkGetPhysicalDeviceSurfacePresentModes2EXT`
- `vkReleaseFullScreenExclusiveModeEXT`

If `VK_KHR_device_group` is supported:

- `vkGetDeviceGroupSurfacePresentModes2EXT`

If `Version 1.1` is supported:

- `vkGetDeviceGroupSurfacePresentModes2EXT`

New Structures

- Extending `VkPhysicalDeviceSurfaceInfo2KHR`, `VkSwapchainCreateInfoKHR`
- `VkSurfaceFullScreenExclusiveInfoEXT`
- Extending `VkSurfaceCapabilities2KHR`:
  - `VkSurfaceCapabilitiesFullScreenExclusiveEXT`

If `VK_KHR_win32_surface` is supported:

- Extending `VkPhysicalDeviceSurfaceInfo2KHR, VkSwapchainCreateInfoKHR`:
  - `VkSurfaceFullScreenExclusiveWin32InfoEXT`

**New Enums**

- `VkFullScreenExclusiveEXT`

**New Enum Constants**

- `VK_EXT_FULL_SCREEN_EXCLUSIVE_EXTENSION_NAME`
- `VK_EXT_FULL_SCREEN_EXCLUSIVE_SPEC_VERSION`
- Extending `VkResult`:
  - `VK_ERROR_FULL_SCREEN_EXCLUSIVE_MODE_LOST_EXT`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_SURFACE_CAPABILITIES_FULL_SCREEN_EXCLUSIVE_EXT`
  - `VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_INFO_EXT`

If `VK_KHR_win32_surface` is supported:

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_SURFACE_FULL_SCREEN_EXCLUSIVE_WIN32_INFO_EXT`

**Issues**

1) What should the extension & flag be called?

**RESOLVED:** `VK_EXT_full_screen-exclusive`.

Other options considered (prior to the app-controlled mode) were:

- `VK_EXT_smooth_fullscreen_transition`
- `VK_EXT_fullscreen_behavior`
- `VK_EXT_fullscreen_preference`
- `VK_EXT_fullscreen_hint`
- `VK_EXT_fast_fullscreen_transition`
- `VK_EXT_avoid_fullscreen_exclusive`

2) Do we need more than a boolean toggle?
RESOLVED: Yes.

Using an enum with default/allowed/disallowed/app-controlled enables applications to accept driver default behavior, specifically override it in either direction without implying the driver is ever required to use full-screen exclusive mechanisms, or manage this mode explicitly.

3) Should this be a KHR or EXT extension?

RESOLVED: EXT, in order to allow it to be shipped faster.

4) Can the fullscreen hint affect the surface capabilities, and if so, should the hint also be specified as input when querying the surface capabilities?

RESOLVED: Yes on both accounts.

While the hint does not guarantee a particular full-screen mode will be used when the swapchain is created, it can sometimes imply particular modes will NOT be used. If the driver determines that it will opt-out of using a particular mode based on the policy, and knows it can only support certain capabilities if that mode is used, it would be confusing at best to the application to report those capabilities in such cases. Not allowing implementations to report this state to applications could result in situations where applications are unable to determine why swapchain creation fails when they specify certain hint values, which could result in never-terminating surface creation loops.

5) Should full-screen be one word or two?

RESOLVED: Two words.

"Fullscreen" is not in my dictionary, and web searches did not turn up definitive proof that it is a colloquially accepted compound word. Documentation for the corresponding Windows API mechanisms dithers. The text consistently uses a hyphen, but none-the-less, there is a SetFullScreenState method in the DXGI swapchain object. Given this inconclusive external guidance, it is best to adhere to the Vulkan style guidelines and avoid inventing new compound words.

Version History

- Revision 4, 2019-03-12 (Tobias Hector)
  - Added application-controlled mode, and related functions
  - Tidied up appendix
- Revision 3, 2019-01-03 (James Jones)
  - Renamed to VK_EXT_full_screen_exclusive
  - Made related adjustments to the tri-state enumerant names.
- Revision 2, 2018-11-27 (James Jones)
  - Renamed to VK_KHR_fullscreen_behavior
  - Switched from boolean flag to tri-state enum
- Revision 1, 2018-11-06 (James Jones)
VK_EXT_global_priority

Name String
VK_EXT_global_priority

Extension Type
Device extension

Registered Extension Number
175

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Andres Rodriguez lostgoat

Other Extension Metadata

Last Modified Date
2017-10-06

IP Status
No known IP claims.

Contributors
• Andres Rodriguez, Valve
• Pierre-Loup Griffais, Valve
• Dan Ginsburg, Valve
• Mitch Singer, AMD

Description
In Vulkan, users can specify device-scope queue priorities. In some cases it may be useful to extend this concept to a system-wide scope. This extension provides a mechanism for caller's to set their system-wide priority. The default queue priority is VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT.

The driver implementation will attempt to skew hardware resource allocation in favour of the higher-priority task. Therefore, higher-priority work may retain similar latency and throughput characteristics even if the system is congested with lower priority work.

The global priority level of a queue shall take precedence over the per-process queue priority
Abuse of this feature may result in starving the rest of the system from hardware resources. Therefore, the driver implementation may deny requests to acquire a priority above the default priority (VK_QUEUE_GLOBAL_PRIORITY_MEDIUM_EXT) if the caller does not have sufficient privileges. In this scenario VK_ERROR_NOT_PERMITTED_EXT is returned.

The driver implementation may fail the queue allocation request if resources required to complete the operation have been exhausted (either by the same process or a different process). In this scenario VK_ERROR_INITIALIZATION_FAILED is returned.

**New Structures**

- Extending VkDeviceQueueCreateInfo:
  - VkDeviceQueueGlobalPriorityCreateInfoEXT

**New Enums**

- VkQueueGlobalPriorityEXT

**New Enum Constants**

- VK_EXT_GLOBAL_PRIORITY_EXTENSION_NAME
- VK_EXT_GLOBAL_PRIORITY_SPEC_VERSION

- Extending VkResult:
  - VK_ERROR_NOT_PERMITTED_EXT

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DEVICE_QUEUE_GLOBAL_PRIORITY_CREATE_INFO_EXT

**Version History**

- Revision 2, 2017-11-03 (Andres Rodriguez)
  - Fixed VkQueueGlobalPriorityEXT missing _EXT suffix
- Revision 1, 2017-10-06 (Andres Rodriguez)
  - First version.

**VK_EXT_hdr_metadata**

**Name String**

VK_EXT_hdr_metadata

**Extension Type**

- Device extension
This extension defines two new structures and a function to assign SMPTE (the Society of Motion Picture and Television Engineers) 2086 metadata and CTA (Consumer Technology Association) 861.3 metadata to a swapchain. The metadata includes the color primaries, white point, and luminance range of the reference monitor, which all together define the color volume that contains all the possible colors the reference monitor can produce. The reference monitor is the display where creative work is done and creative intent is established. To preserve such creative intent as much as possible and achieve consistent color reproduction on different viewing displays, it is useful for the display pipeline to know the color volume of the original reference monitor where content was created or tuned. This avoids performing unnecessary mapping of colors that are not displayable on the original reference monitor. The metadata also includes the maxContentLightLevel and maxFrameAverageLightLevel as defined by CTA 861.3.

While the general purpose of the metadata is to assist in the transformation between different color volumes of different displays and help achieve better color reproduction, it is not in the scope of this extension to define how exactly the metadata should be used in such a process. It is up to the implementation to determine how to make use of the metadata.

**New Commands**

- `vkSetHdrMetadataEXT`
New Structures

- VkHdrMetadataEXT
- VkXYColorEXT

New Enum Constants

- VK_EXT_HDR_METADATA_EXTENSION_NAME
- VK_EXT_HDR_METADATA_SPEC_VERSION

Extending VkStructureType:
- VK_STRUCTURE_TYPE_HDR_METADATA_EXT

Issues

1) Do we need a query function?

**PROPOSED**: No, Vulkan does not provide queries for state that the application can track on its own.

2) Should we specify default if not specified by the application?

**PROPOSED**: No, that leaves the default up to the display.

Version History

- Revision 1, 2016-12-27 (Courtney Goeltzenleuchter)
  - Initial version
- Revision 2, 2018-12-19 (Courtney Goeltzenleuchter)
  - Correct implicit validity for VkHdrMetadataEXT structure

**VK_EXT_headless_surface**

Name String

VK_EXT_headless_surface

Extension Type

Instance extension

Registered Extension Number

257

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_surface
Description

The VK_EXT_headless_surface extension is an instance extension. It provides a mechanism to create VkSurfaceKHR objects independently of any window system or display device. The presentation operation for a swapchain created from a headless surface is by default a no-op, resulting in no externally-visible result.

Because there is no real presentation target, future extensions can layer on top of the headless surface to introduce arbitrary or customisable sets of restrictions or features. These could include features like saving to a file or restrictions to emulate a particular presentation target.

This functionality is expected to be useful for application and driver development because it allows any platform to expose an arbitrary or customisable set of restrictions and features of a presentation engine. This makes it a useful portable test target for applications targeting a wide range of presentation engines where the actual target presentation engines might be scarce, unavailable or otherwise undesirable or inconvenient to use for general Vulkan application development.

New Commands

- vkCreateHeadlessSurfaceEXT

New Structures

- VkHeadlessSurfaceCreateInfoEXT

New Bitmasks

- VkHeadlessSurfaceCreateFlagsEXT

New Enum Constants

- VK_EXT_HEADLESS_SURFACE_EXTENSION_NAME
- VK_EXT_HEADLESS_SURFACE_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_HEADLESS_SURFACE_CREATE_INFO_EXT

Version History

• Revision 1, 2019-03-21 (Ray Smith)
  ◦ Initial draft

VK_EXT_host_query_reset

Name String

VK_EXT_host_query_reset

Extension Type

Device extension

Registered Extension Number

262

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
  • Requires VK_KHR_get_physical_device_properties2

Deprecation state

• Promoted to Vulkan 1.2

Contact

• Bas Nieuwenhuizen BNieuwenhuizen

Other Extension Metadata

Last Modified Date

2019-03-06

IP Status

No known IP claims.

Interactions and External Dependencies

• Promoted to Vulkan 1.2 Core

Contributors

• Bas Nieuwenhuizen, Google
  • Jason Ekstrand, Intel
Description

This extension adds a new function to reset queries from the host.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2, with the EXT suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- vkResetQueryPoolEXT

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceHostQueryResetFeaturesEXT

New Enum Constants

- VK_EXT_HOST_QUERY_RESET_EXTENSION_NAME
- VK_EXT_HOST_QUERY_RESET_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_HOST_QUERY_RESET_FEATURES_EXT

Version History

- Revision 1, 2019-03-12 (Bas Nieuwenhuizen)
  - Initial draft

VK_EXT_image_drm_format_modifier

Name String

VK_EXT_image_drm_format_modifier

Extension Type

Device extension

Registered Extension Number

159

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_bind_memory2
- Requires VK_KHR_get_physical_device_properties2
- Requires VK_KHR_image_format_list
- Requires VK_KHR_sampler_yCbCr_conversion

Contact

- Chad Versace chadversary

Other Extension Metadata

Last Modified Date

2018-08-29

IP Status

No known IP claims.

Contributors

- Antoine Labour, Google
- Bas Nieuwenhuizen, Google
- Chad Versace, Google
- James Jones, NVIDIA
- Jason Ekstrand, Intel
- Jörg Wagner, ARM
- Kristian Høgsberg Kristensen, Google
- Ray Smith, ARM

Description

This extension provides the ability to use DRM format modifiers with images, enabling Vulkan to better integrate with the Linux ecosystem of graphics, video, and display APIs.

Its functionality closely overlaps with EGL_EXT_image_dma_buf_import_modifiers\(^2\) and EGL_MESA_image_dma_buf_export\(^3\). Unlike the EGL extensions, this extension does not require the use of a specific handle type (such as a dma_buf) for external memory and provides more explicit control of image creation.

Introduction to DRM Format Modifiers

A DRM format modifier is a 64-bit, vendor-prefixed, semi-opaque unsigned integer. Most modifiers represent a concrete, vendor-specific tiling format for images. Some exceptions are DRM_FORMAT_MOD_LINEAR (which is not vendor-specific); DRM_FORMAT_MOD_NONE (which is an alias of DRM_FORMAT_MOD_LINEAR due to historical accident); and DRM_FORMAT_MOD_INVALID (which does not
represent a tiling format). The modifier’s vendor prefix consists of the 8 most significant bits. The canonical list of modifiers and vendor prefixes is found in `drm_fourcc.h` in the Linux kernel source. The other dominant source of modifiers are vendor kernel trees.

One goal of modifiers in the Linux ecosystem is to enumerate for each vendor a reasonably sized set of tiling formats that are appropriate for images shared across processes, APIs, and/or devices, where each participating component may possibly be from different vendors. A non-goal is to enumerate all tiling formats supported by all vendors. Some tiling formats used internally by vendors are inappropriate for sharing; no modifiers should be assigned to such tiling formats.

Modifier values typically do not describe memory layouts. More precisely, a modifier’s lower 56 bits usually have no structure. Instead, modifiers name memory layouts; they name a small set of vendor-preferred layouts for image sharing. As a consequence, in each vendor namespace the modifier values are often sequentially allocated starting at 1.

Each modifier is usually supported by a single vendor and its name matches the pattern `{VENDOR}_FORMAT_MOD_*` or `DRM_FORMAT_MOD_{VENDOR}_*`. Examples are `I915_FORMAT_MOD_X_TILED` and `DRM_FORMAT_MOD_BROADCOM_VC4_T_TILED`. An exception is `DRM_FORMAT_MOD_LINEAR`, which is supported by most vendors.

Many APIs in Linux use modifiers to negotiate and specify the memory layout of shared images. For example, a Wayland compositor and Wayland client may, by relaying modifiers over the Wayland protocol `zwp_linux_dmabuf_v1`, negotiate a vendor-specific tiling format for a shared `wl_buffer`. The client may allocate the underlying memory for the `wl_buffer` with GBM, providing the chosen modifier to `gbm_bo_create_withModifiers`. The client may then import the `wl_buffer` into Vulkan for producing image content, providing the resource’s `dma_buf` to `VkImportMemoryFdInfoKHR` and its modifier to `VkImageDrmFormatModifierExplicitCreateInfoEXT`. The compositor may then import the `wl_buffer` into OpenGL for sampling, providing the resource’s `dma_buf` and modifier to `eglCreateImage`. The compositor may also bypass OpenGL and submit the `wl_buffer` directly to the kernel’s display API, providing the `dma_buf` and modifier through `drm_mode_fb_cmd2`.

**Format Translation**

Modifier-capable APIs often pair modifiers with DRM formats, which are defined in `drm_fourcc.h`. However, `VK_EXT_image_drm_format_modifier` uses `VkFormat` instead of DRM formats. The application must convert between `VkFormat` and DRM format when it sends or receives a DRM format to or from an external API.

The mapping from `VkFormat` to DRM format is lossy. Therefore, when receiving a DRM format from an external API, often the application must use information from the external API to accurately map the DRM format to a `VkFormat`. For example, DRM formats do not distinguish between RGB and sRGB (as of 2018-03-28); external information is required to identify the image’s colorspace.

The mapping between `VkFormat` and DRM format is also incomplete. For some DRM formats there exist no corresponding Vulkan format, and for some Vulkan formats there exist no corresponding DRM format.
Usage Patterns

Three primary usage patterns are intended for this extension:

• **Negotiation.** The application negotiates with *modifier*-aware, external components to determine sets of image creation parameters supported among all components.

In the Linux ecosystem, the negotiation usually assumes the image is a 2D, single-sampled, non-mipmapped, non-array image; this extension permits that assumption but does not require it. The result of the negotiation usually resembles a set of tuples such as $(\text{drmFormat, drmFormatModifier})$, where each participating component supports all tuples in the set.

Many details of this negotiation—such as the protocol used during negotiation, the set of image creation parameters expressable in the protocol, and how the protocol chooses which process and which API will create the image—are outside the scope of this specification.

In this extension, `vkGetPhysicalDeviceFormatProperties2` with `VkDrmFormatModifierPropertiesListEXT` serves a primary role during the negotiation, and `vkGetPhysicalDeviceImageFormatProperties2` with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT` serves a secondary role.

• **Import.** The application imports an image with a *modifier*.

In this pattern, the application receives from an external source the image's memory and its creation parameters, which are often the result of the negotiation described above. Some image creation parameters are implicitly defined by the external source; for example, `VK_IMAGE_TYPE_2D` is often assumed. Some image creation parameters are usually explicit, such as the image's *format*, *drmFormatModifier*, and *extent*; and each plane's *offset* and *rowPitch*.

Before creating the image, the application first verifies that the physical device supports the received creation parameters by querying `vkGetPhysicalDeviceFormatProperties2` with `VkDrmFormatModifierPropertiesListEXT` and `vkGetPhysicalDeviceImageFormatProperties2` with `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`. Then the application creates the image by chaining `VkImageDrmFormatModifierExplicitCreateInfoEXT` and `VkExternalMemoryImageCreateInfo` onto `VkImageCreateInfo`.

• **Export.** The application creates an image and allocates its memory. Then the application exports to *modifier*-aware consumers the image's memory handles; its creation parameters; its *modifier*; and the *offset*, *size*, and *rowPitch* of each *memory plane*.

In this pattern, the Vulkan device is the authority for the image; it is the allocator of the image's memory and the decider of the image's creation parameters. When choosing the image's creation parameters, the application usually chooses a tuple $(\text{format, drmFormatModifier})$ from the result of the negotiation described above. The negotiation's result often contains multiple tuples that share the same format but differ in their *modifier*. In this case, the application should defer the choice of the image's *modifier* to the Vulkan implementation by providing all such *modifiers* to `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`; and the implementation should choose from `pDrmFormatModifiers` the optimal *modifier* in consideration with the other image parameters.
The application creates the image by chaining `VkImageDrmFormatModifierListCreateInfoEXT` and `VkExternalMemoryImageCreateInfo` onto `VkImageCreateInfo`. The protocol and APIs by which the application will share the image with external consumers will likely determine the value of `VkExternalMemoryImageCreateInfo::handleTypes`. The implementation chooses for the image an optimal modifier from `VkImageDrmFormatModifierListCreateInfoEXT::pDrmFormatModifiers`. The application then queries the implementation-chosen modifier with `vkGetImageDrmFormatModifierPropertiesEXT`, and queries the memory layout of each plane with `vkGetImageSubresourceLayout`.

The application then allocates the image's memory with `VkMemoryAllocateInfo`, adding chained extending structures for external memory; binds it to the image; and exports the memory, for example, with `vkGetMemoryFdKHR`.

Finally, the application sends the image's creation parameters, its modifier, its per-plane memory layout, and the exported memory handle to the external consumers. The details of how the application transmits this information to external consumers is outside the scope of this specification.

### Prior Art

Extension `EGL_EXT_image_dma_buf_import` introduced the ability to create an `EGLImage` by importing for each plane a dma_buf, offset, and row pitch.

Later, extension `EGL_EXT_image_dma_buf_import_modifiers` introduced the ability to query which combination of formats and modifiers the implementation supports and to specify modifiers during creation of the `EGLImage`.

Extension `EGL_MESA_image_dma_buf_export` is the inverse of `EGL_EXT_image_dma_buf_import_modifiers`.

The Linux kernel modesetting API (KMS), when configuring the display's framebuffer with `struct drm_mode_fb_cmd2`, allows one to specify the framebuffer's modifier as well as a per-plane memory handle, offset, and row pitch.

GBM, a graphics buffer manager for Linux, allows creation of a `gbm_bo` (that is, a graphics buffer object) by importing data similar to that in `EGL_EXT_image_dma_buf_import_modifiers`; and symmetrically allows exporting the same data from the `gbm_bo`. See the references to modifier and plane in `gbm.h`.

### New Commands

- `vkGetImageDrmFormatModifierPropertiesEXT`

### New Structures

- `VkDrmFormatModifierPropertiesEXT`
- `VkImageDrmFormatModifierPropertiesEXT`
- Extending `VkFormatProperties2`:
  - `VkDrmFormatModifierPropertiesListEXT`
• Extending `VkImageCreateInfo`:
  ◦ `VkImageDrmFormatModifierExplicitCreateInfoEXT`
  ◦ `VkImageDrmFormatModifierListCreateInfoEXT`

• Extending `VkPhysicalDeviceImageFormatInfo2`:
  ◦ `VkPhysicalDeviceImageDrmFormatModifierInfoEXT`

**New Enum Constants**

• `VK_EXT_IMAGE_DRM_FORMAT_MODIFIER_EXTENSION_NAME`
• `VK_EXT_IMAGE_DRM_FORMAT_MODIFIER_SPEC_VERSION`

• Extending `VkImageAspectFlagBits`:
  ◦ `VK_IMAGE_ASPECT_MEMORY_PLANE_0_BIT_EXT`
  ◦ `VK_IMAGE_ASPECT_MEMORY_PLANE_1_BIT_EXT`
  ◦ `VK_IMAGE_ASPECT_MEMORY_PLANE_2_BIT_EXT`
  ◦ `VK_IMAGE_ASPECT_MEMORY_PLANE_3_BIT_EXT`

• Extending `VkImageTiling`:
  ◦ `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`

• Extending `VkResult`:
  ◦ `VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_DRM_FORMAT_MODIFIER_PROPERTIES_LIST_EXT`
  ◦ `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_EXPLICIT_CREATE_INFO_EXT`
  ◦ `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_LIST_CREATE_INFO_EXT`
  ◦ `VK_STRUCTURE_TYPE_IMAGE_DRM_FORMAT_MODIFIER_PROPERTIES_EXT`
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_DRM_FORMAT_MODIFIER_INFO_EXT`

**Issues**

1) Should this extension define a single DRM format modifier per `VkImage`? Or define one per plane?

+ [RESOLVED: There exists a single DRM format modifier per `VkImage`.]

**DISCUSSION:** Prior art, such as `EGL_EXT_image_dma_buf_import_modifiers`, `struct drm_mode_fb_cmd2`, and `struct gbm_import_fd_modifier_data`, allows defining one `modifier` per plane. However, developers of the GBM and kernel APIs concede it was a mistake. Beginning in Linux 4.10, the kernel requires that the application provide the same DRM format `modifier` for each plane. (See Linux commit `bae781b259269590109e8a4a8227331362b88212`). And GBM provides an entry point, `gbm_bo_get_modifier`, for querying the `modifier` of the image but does not provide one to query the modifier of individual planes.
2) When creating an image with `VkImageDrmFormatModifierExplicitCreateInfoEXT`, which is typically used when importing an image, should the application explicitly provide the size of each plane?

+  

**RESOLVED:** No. The application must not provide the size. To enforce this, the API requires that `VkImageDrmFormatModifierExplicitCreateInfoEXT::pPlaneLayouts->size` must be 0.

**DISCUSSION:** Prior art, such as `EGL_EXT_image_dma_buf_import_modifiers`\(^2\), `struct drm_mode_fb_cmd2`\(^4\), and `struct gbm_import_fd_modifier_data`\(^5\), omits from the API the size of each plane. Instead, the APIs infer each plane’s size from the import parameters, which include the image’s pixel format and a dma_buf, offset, and row pitch for each plane.

However, Vulkan differs from EGL and GBM with regards to image creation in the following ways:

**Differences in Image Creation**

- **Undedicated allocation by default.** When importing or exporting a set of dma_bufs as an `EGLImage` or `gbm_bo`, common practice mandates that each dma_buf’s memory be dedicated (in the sense of `VK_KHR_dedicated_allocation`) to the image (though not necessarily dedicated to a single plane). In particular, neither the GBM documentation nor the EGL extension specifications explicitly state this requirement, but in light of common practice this is likely due to under-specification rather than intentional omission. In contrast, `VK_EXT_image_drm_format_modifier` permits, but does not require, the implementation to require dedicated allocations for images created with `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT`.

- **Separation of image creation and memory allocation.** When importing a set of dma_bufs as an `EGLImage` or `gbm_bo`, EGL and GBM create the image resource and bind it to memory (the dma_bufs) simultaneously. This allows EGL and GBM to query each dma_buf’s size during image creation. In Vulkan, image creation and memory allocation are independent unless a dedicated allocation is used (as in `VK_KHR_dedicated_allocation`). Therefore, without requiring dedicated allocation, Vulkan cannot query the size of each dma_buf (or other external handle) when calculating the image’s memory layout. Even if dedication allocation were required, Vulkan cannot calculate the image’s memory layout until after the image is bound to its dma_ufs.

The above differences complicate the potential inference of plane size in Vulkan. Consider the following problematic cases:

**Problematic Plane Size Calculations**

- **Padding.** Some plane of the image may require implementation-dependent padding.

- **Metadata.** For some modifiers, the image may have a metadata plane which requires a non-trivial calculation to determine its size.

- **Mipmapped, array, and 3D images.** The implementation may support `VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT` for images whose `mipLevels`, `arrayLayers`, or `depth` is greater than 1. For such images with certain modifiers, the calculation of each plane’s size may be non-trivial.

However, an application-provided plane size solves none of the above problems.
For simplicity, consider an external image with a single memory plane. The implementation is obviously capable calculating the image’s size when its tiling is VK_IMAGE_TILING_OPTIMAL. Likewise, any reasonable implementation is capable of calculating the image’s size when its tiling uses a supported modifier.

Suppose that the external image’s size is smaller than the implementation-calculated size. If the application provided the external image’s size to vkCreateImage, the implementation would observe the mismatched size and recognize its inability to comprehend the external image’s layout (unless the implementation used the application-provided size to select a refinement of the tiling layout indicated by the modifier, which is strongly discouraged). The implementation would observe the conflict, and reject image creation with VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT. On the other hand, if the application did not provide the external image’s size to vkCreateImage, then the application would observe after calling vkGetImageMemoryRequirements that the external image’s size is less than the size required by the implementation. The application would observe the conflict and refuse to bind the VkImage to the external memory. In both cases, the result is explicit failure.

Suppose that the external image’s size is larger than the implementation-calculated size. If the application provided the external image’s size to vkCreateImage, for reasons similar to above the implementation would observe the mismatched size and recognize its inability to comprehend the image data residing in the extra size. The implementation, however, must assume that image data resides in the entire size provided by the application. The implementation would observe the conflict and reject image creation with VK_ERROR_INVALID_DRM_FORMAT_MODIFIER_PLANE_LAYOUT_EXT. On the other hand, if the application did not provide the external image’s size to vkCreateImage, then the application would observe after calling vkGetImageMemoryRequirements that the external image’s size is larger than the implementation-usable size. The application would observe the conflict and refuse to bind the VkImage to the external memory. In both cases, the result is explicit failure.

Therefore, an application-provided size provides no benefit, and this extension should not require it. This decision renders VkSubresourceLayout::size an unused field during image creation, and thus introduces a risk that implementations may require applications to submit sideband creation parameters in the unused field. To prevent implementations from relying on sideband data, this extension requires the application to set size to 0.

References

1. EGL_EXT_image_dma_buf_import
2. EGL_EXT_image_dma_buf_import_modifiers
3. EGL_MESA_image_dma_buf_export
4. struct drm_mode_fb_cmd2
5. gbm.h

Version History

• Revision 1, 2018-08-29 (Chad Versace)
  ◦ First stable revision
VK_EXT_image_robustness

Name String
VK_EXT_image_robustness

Extension Type
Device extension

Registered Extension Number
336

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact
• Graeme Leese @gnl21

Other Extension Metadata

Last Modified Date
2020-04-27

IP Status
No known IP claims.

Contributors
• Graeme Leese, Broadcom
• Jan-Harald Fredriksen, ARM
• Jeff Bolz, NVIDIA
• Spencer Fricke, Samsung
• Courtney Goeltzenleuchter, Google
• Slawomir Cygan, Intel

Description
This extension adds stricter requirements for how out of bounds reads from images are handled. Rather than returning undefined values, most out of bounds reads return R, G, and B values of zero and alpha values of either zero or one. Components not present in the image format may be set to zero or to values based on the format as described in Conversion to RGBA.
New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceImageRobustnessFeaturesEXT

New Enum Constants

- VK_EXT_IMAGE_ROBUSTNESS_EXTENSION_NAME
- VK_EXT_IMAGE_ROBUSTNESS_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_ROBUSTNESS_FEATURES_EXT

Issues

1. How does this extension differ from VK_EXT_robustness2?

The guarantees provided by this extension are a subset of those provided by the robustImageAccess2 feature of VK_EXT_robustness2. Where this extension allows return values of (0, 0, 0, 0) or (0, 0, 0, 1), robustImageAccess2 requires that a particular value dependent on the image format be returned. This extension provides no guarantees about the values returned for an access to an invalid Lod.

Examples

None.

Version History

- Revision 1, 2020-04-27 (Graeme Leese)
  - Initial draft

VK_EXT_index_type_uint8

Name String
  VK_EXT_index_type_uint8

Extension Type
  Device extension

Registered Extension Number
  266

Revision
  1

Extension and Version Dependencies
  - Requires Vulkan 1.0
Contact

- Piers Daniell

Other Extension Metadata

Last Modified Date
- 2019-05-02

IP Status
- No known IP claims.

Contributors
- Jeff Bolz, NVIDIA

Description

This extension allows uint8_t indices to be used with vkCmdBindIndexBuffer.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceIndexTypeUint8FeaturesEXT

New Enum Constants

- VK_EXT_INDEX_TYPE_UINT8_EXTENSION_NAME
- VK_EXT_INDEX_TYPE_UINT8_SPEC_VERSION
- Extending VkIndexType:
  - VK_INDEX_TYPE_UINT8_EXT
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INDEX_TYPE_UINT8_FEATURES_EXT

Version History

- Revision 1, 2019-05-02 (Piers Daniell)
  - Internal revisions

VK_EXT_inline_uniform_block

Name String

- VK_EXT_inline_uniform_block

Extension Type

- Device extension
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2
- Requires VK_KHR_maintenance1

Contact

- Daniel Rakos @aqnuep

Other Extension Metadata

Last Modified Date

2018-08-01

IP Status

No known IP claims.

Contributors

- Daniel Rakos, AMD
- Jeff Bolz, NVIDIA
- Slawomir Grajewski, Intel
- Neil Henning, Codeplay

Description

This extension introduces the ability to back uniform blocks directly with descriptor sets by storing inline uniform data within descriptor pool storage. Compared to push constants this new construct allows uniform data to be reused across multiple disjoint sets of draw or dispatch commands and may enable uniform data to be accessed with less indirections compared to uniforms backed by buffer memory.

New Structures

- Extending VkDescriptorPoolCreateInfo:
  - VkDescriptorPoolInlineUniformBlockCreateInfoEXT
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceInlineUniformBlockFeaturesEXT
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceInlineUniformBlockPropertiesEXT
• Extending VkWriteDescriptorSet:
  ◦ VkWriteDescriptorSetInlineUniformBlockEXT

New Enum Constants

• VK_EXT_INLINE_UNIFORM_BLOCK_EXTENSION_NAME
• VK_EXT_INLINE_UNIFORM_BLOCK_SPEC_VERSION

• Extending VkDescriptorType:
  ◦ VK_DESCRIPTOR_TYPE_INLINE_UNIFORM_BLOCK_EXT

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_INLINE_UNIFORM_BLOCK_CREATE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INLINE_UNIFORM_BLOCK_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_INLINE_UNIFORM_BLOCK_EXT

Issues

1) Do we need a new storage class for inline uniform blocks vs uniform blocks?

RESOLVED: No. The Uniform storage class is used to allow the same syntax used for both uniform buffers and inline uniform blocks.

2) Is the descriptor array index and array size expressed in terms of bytes or dwords for inline uniform block descriptors?

RESOLVED: In bytes, but both must be a multiple of 4, similar to how push constant ranges are specified. The descriptorCount of VkDescriptorSetLayoutBinding thus provides the total number of bytes a particular binding with an inline uniform block descriptor type can hold, while the srcArrayElement, dstArrayElement, and descriptorCount members of VkWriteDescriptorSet, VkCopyDescriptorSet, and VkDescriptorUpdateTemplateEntry (where applicable) specify the byte offset and number of bytes to write/copy to the binding’s backing store. Additionally, the stride member of VkDescriptorUpdateTemplateEntry is ignored for inline uniform blocks and a default value of one is used, meaning that the data to update inline uniform block bindings with must be contiguous in memory.

3) What layout rules apply for uniform blocks corresponding to inline constants?

RESOLVED: They use the same layout rules as uniform buffers.

4) Do we need to add non-uniform indexing features/properties as introduced by VK_EXT_descriptor_indexing for inline uniform blocks?

RESOLVED: No, because inline uniform blocks are not allowed to be “arrayed”. A single binding with an inline uniform block descriptor type corresponds to a single uniform block instance and the array indices inside that binding refer to individual offsets within the uniform block (see issue #2). However, this extension does introduce new features/properties about the level of support for...
update-after-bind inline uniform blocks.

5) Is the `descriptorBindingVariableDescriptorCount` feature introduced by `VK_EXT_descriptor_indexing` supported for inline uniform blocks?

**RESOLVED**: Yes, as long as other inline uniform block specific limits are respected.

6) Do the robustness guarantees of `robustBufferAccess` apply to inline uniform block accesses?

**RESOLVED**: No, similarly to push constants, as they are not backed by buffer memory like uniform buffers.

**Version History**

- Revision 1, 2018-08-01 (Daniel Rakos)
  - Internal revisions

**VK_EXT_line_rasterization**

**Name String**

VK_EXT_line_rasterization

**Extension Type**

Device extension

**Registered Extension Number**

260

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

**Special Use**

- CAD support

**Contact**

- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

**Other Extension Metadata**

**Last Modified Date**

2019-05-09

**IP Status**

No known IP claims.
Contributors

• Jeff Bolz, NVIDIA
• Allen Jensen, NVIDIA
• Jason Ekstrand, Intel

Description

This extension adds some line rasterization features that are commonly used in CAD applications and supported in other APIs like OpenGL. Bresenham-style line rasterization is supported, smooth rectangular lines (coverage to alpha) are supported, and stippled lines are supported for all three line rasterization modes.

New Commands

• vkCmdSetLineStippleEXT

New Structures

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceLineRasterizationFeaturesEXT
• Extending VkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceLineRasterizationPropertiesEXT
• Extending VkPipelineRasterizationStateCreateInfo:
  ◦ VkPipelineRasterizationLineStateCreateInfoEXT

New Enums

• VkLineRasterizationModeEXT

New Enum Constants

• VK_EXT_LINE_RASTERIZATION_EXTENSION_NAME
• VK_EXT_LINE_RASTERIZATION_SPEC_VERSION
• Extending VkDynamicState:
  ◦ VK_DYNAMIC_STATE_LINE_STIPPLE_EXT
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_LINE_RASTERIZATION_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_LINE_STATE_CREATE_INFO_EXT

Issues
Do we need to support Bresenham-style and smooth lines with more than one rasterization sample? i.e. the equivalent of `glDisable(GL_MULTISAMPLE)` in OpenGL when the framebuffer has more than one sample?

RESOLVED: Yes.
For simplicity, Bresenham line rasterization carries forward a few restrictions from OpenGL, such as not supporting per-sample shading, alpha to coverage, or alpha to one.

**Version History**

- Revision 1, 2019-05-09 (Jeff Bolz)
  - Initial draft

**VK_EXT_memory_budget**

**Name String**

VK_EXT_memory_budget

**Extension Type**

Device extension

**Registered Extension Number**

238

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Jeff Bolz @jeffbolznv

**Other Extension Metadata**

**Last Modified Date**

2018-10-08

**Contributors**

- Jeff Bolz, NVIDIA
- Jeff Juliano, NVIDIA
Description

While running a Vulkan application, other processes on the machine might also be attempting to use the same device memory, which can pose problems. This extension adds support for querying the amount of memory used and the total memory budget for a memory heap. The values returned by this query are implementation-dependent and can depend on a variety of factors including operating system and system load.

The `VkPhysicalDeviceMemoryBudgetPropertiesEXT::heapBudget` values can be used as a guideline for how much total memory from each heap the current process can use at any given time, before allocations may start failing or causing performance degradation. The values may change based on other activity in the system that is outside the scope and control of the Vulkan implementation.

The `VkPhysicalDeviceMemoryBudgetPropertiesEXT::heapUsage` will display the current process estimated heap usage.

With this information, the idea is for an application at some interval (once per frame, per few seconds, etc) to query `heapBudget` and `heapUsage`. From here the application can notice if it is over budget and decide how it wants to handle the memory situation (free it, move to host memory, changing mipmap levels, etc). This extension is designed to be used in concert with `VK_EXT_memory_priority` to help with this part of memory management.

New Structures

- Extending `VkPhysicalDeviceMemoryProperties2`:
  - `VkPhysicalDeviceMemoryBudgetPropertiesEXT`

New Enum Constants

- `VK_EXT_MEMORY_BUDGET_EXTENSION_NAME`
- `VK_EXT_MEMORY_BUDGET_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_BUDGET_PROPERTIES_EXT`

Version History

- Revision 1, 2018-10-08 (Jeff Bolz)
  - Initial revision

VK_EXT_memory_priority

Name String

- `VK_EXT_memory_priority`

Extension Type

- Device extension
Description

This extension adds a priority value specified at memory allocation time. On some systems with both device-local and non-device-local memory heaps, the implementation may transparently move memory from one heap to another when a heap becomes full (for example, when the total memory used across all processes exceeds the size of the heap). In such a case, this priority value may be used to determine which allocations are more likely to remain in device-local memory.

New Structures

• Extending VkMemoryAllocateInfo:
  ◦ VkMemoryPriorityAllocateInfoEXT

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceMemoryPriorityFeaturesEXT

New Enum Constants

• VK_EXT_MEMORY_PRIORITY_EXTENSION_NAME
• VK_EXT_MEMORY_PRIORITY_SPEC_VERSION

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_MEMORY_PRIORITY_ALLOCATE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PRIORITY_FEATURES_EXT
Version History

• Revision 1, 2018-10-08 (Jeff Bolz)
  ◦ Initial revision

VK_EXT_metal_surface

Name String

VK_EXT_metal_surface

Extension Type

Instance extension

Registered Extension Number

218

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_surface

Contact

• Dzmitry Malyshau kvark

Other Extension Metadata

Last Modified Date

2018-10-01

IP Status

No known IP claims.

Contributors

• Dzmitry Malyshau, Mozilla Corp.

Description

The VK_EXT_metal_surface extension is an instance extension. It provides a mechanism to create a VkSurfaceKHR object (defined by the VK_KHR_surface extension) from CAMetalLayer, which is the native rendering surface of Apple’s Metal framework.

New Base Types

• CAMetalLayer
New Commands

• `vkCreateMetalSurfaceEXT`

New Structures

• `VkMetalSurfaceCreateInfoEXT`

New Bitmasks

• `VkMetalSurfaceCreateFlagsEXT`

New Enum Constants

• `VK_EXT_METAL_SURFACE_EXTENSION_NAME`
• `VK_EXT_METAL_SURFACE_SPEC_VERSION`
• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_METAL_SURFACE_CREATE_INFO_EXT`

Version History

• Revision 1, 2018-10-01 (Dzmitry Malyshau)
  ◦ Initial version

`VK_EXT_pci_bus_info`

Name String

`VK_EXT_pci_bus_info`

Extension Type

Device extension

Registered Extension Number

213

Revision

2

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires `VK_KHR_get_physical_device_properties2`

Contact

• Matthaeus G. Chajdas @anteru
Other Extension Metadata

Last Modified Date
2018-12-10

IP Status
No known IP claims.

Contributors
- Matthaeus G. Chajdas, AMD
- Daniel Rakos, AMD

Description
This extension adds a new query to obtain PCI bus information about a physical device.

Not all physical devices have PCI bus information, either due to the device not being connected to the system through a PCI interface or due to platform specific restrictions and policies. Thus this extension is only expected to be supported by physical devices which can provide the information.

As a consequence, applications should always check for the presence of the extension string for each individual physical device for which they intend to issue the new query for and should not have any assumptions about the availability of the extension on any given platform.

New Structures
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDevicePCIBusInfoPropertiesEXT`

New Enum Constants
- `VK_EXT_PCI_BUS_INFO_EXTENSION_NAME`
- `VK_EXT_PCI_BUS_INFO_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PCI_BUS_INFO_PROPERTIES_EXT`

Version History
- Revision 2, 2018-12-10 (Daniel Rakos)
  - Changed all members of the new structure to have the uint32_t type
- Revision 1, 2018-10-11 (Daniel Rakos)
  - Initial revision

`VK_EXT_pipeline_creation_cache_control`
Name String

VK_EXT_pipeline_creation_cache_control

Extension Type

Device extension

Registered Extension Number

298

Revision

3

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Gregory Grebe grgrebe_amd

Other Extension Metadata

Last Modified Date

2020-03-23

IP Status

No known IP claims.

Contributors

• Gregory Grebe, AMD
• Tobias Hector, AMD
• Matthaeus Chajdas, AMD
• Mitch Singer, AMD
• Spencer Fricke, Samsung Electronics
• Stuart Smith, Imagination Technologies
• Jeff Bolz, NVIDIA Corporation
• Daniel Koch, NVIDIA Corporation
• Dan Ginsburg, Valve Corporation
• Jeff Leger, QUALCOMM
• Michal Pietraśiuk, Intel
• Jan-Harald Fredriksen, Arm Limited

Description

This extension adds flags to Vk*PipelineCreateInfo and VkPipelineCacheCreateInfo structures with the aim of improving the predictability of pipeline creation cost. The goal is to provide information
about potentially expensive hazards within the client driver during pipeline creation to the application before carrying them out rather than after.

**Background**

Pipeline creation is a costly operation, and the explicit nature of the Vulkan design means that cost is not hidden from the developer. Applications are also expected to schedule, prioritize, and load balance all calls for pipeline creation. It is strongly advised that applications create pipelines sufficiently ahead of their usage. Failure to do so will result in an unresponsive application, intermittent stuttering, or other poor user experiences. Proper usage of pipeline caches and/or derivative pipelines help mitigate this but is not assured to eliminate disruption in all cases. In the event that an ahead-of-time creation is not possible, considerations should be taken to ensure that the current execution context is suitable for the workload of pipeline creation including possible shader compilation.

Applications making API calls to create a pipeline must be prepared for any of the following to occur:

- OS/kernel calls to be made by the ICD
- Internal memory allocation not tracked by the `pAllocator` passed to `vkCreatePipelines`
- Internal thread synchronization or yielding of the current thread’s core
- Extremely long (multi-millisecond+), blocking, compilation times
- Arbitrary call stacks depths and stack memory usage

The job or task based game engines that are being developed to take advantage of explicit graphics APIs like Vulkan may behave exceptionally poorly if any of the above scenarios occur. However, most game engines are already built to “stream” in assets dynamically as the user plays the game. By adding control by way of `VkPipelineCreateFlags`, we can require an ICD to report back a failure in critical execution paths rather than forcing an unexpected wait.

Applications can prevent unexpected compilation by setting `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT` on `Vk*PipelineCreateInfo::flags`. When set, an ICD must not attempt pipeline or shader compilation to create the pipeline object. The ICD will return the result `VK_PIPELINE_COMPILE_REQUIRED_EXT`. An ICD may still return a valid `VkPipeline` object by either re-using existing pre-compiled objects such as those from a pipeline cache, or derivative pipelines.

By default `vkCreatePipelines` calls must attempt to create all pipelines before returning. Setting `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT` on `Vk*PipelineCreateInfo::flags` can be used as an escape hatch for batched pipeline creates.

Hidden locks also add to the unpredictability of the cost of pipeline creation. The most common case of locks inside the `vkCreatePipelines` is internal synchronization of the `VkPipelineCache` object. `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT` can be set when calling `vkCreatePipelineCache` to state the cache is externally synchronized.

The hope is that armed with this information application and engine developers can leverage existing asset streaming systems to recover from "just-in-time" pipeline creation stalls.

---

2770 | Appendix E: Layers & Extensions (Informative)
New Structures

• Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  ◦ `VkPhysicalDevicePipelineCreationCacheControlFeaturesEXT`

New Enum Constants

• `VK_EXT_PIPELINE_CREATION_CACHE_CONTROL_EXTENSION_NAME`
• `VK_EXT_PIPELINE_CREATION_CACHE_CONTROL_SPEC_VERSION`

• Extending `VkPipelineCacheCreateFlagBits`:
  ◦ `VK_PIPELINE_CACHE_CREATE_EXTERNALLY_SYNCHRONIZED_BIT_EXT`

• Extending `VkPipelineCreateFlagBits`:
  ◦ `VK_PIPELINE_CREATE_EARLY_RETURN_ON_FAILURE_BIT_EXT`
  ◦ `VK_PIPELINE_CREATE_FAIL_ON_PIPELINE_COMPILE_REQUIRED_BIT_EXT`

• Extending `VkResult`:
  ◦ `VK_PIPELINE_COMPILE_REQUIRED_EXT`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PIPELINE_CREATION_CACHE_CONTROL_FEATURES_EXT`

Version History

• Revision 1, 2019-11-01 (Gregory Grebe)
  ◦ Initial revision

• Revision 2, 2020-02-24 (Gregory Grebe)
  ◦ Initial public revision

• Revision 3, 2020-03-23 (Tobias Hector)
  ◦ Changed `VK_PIPELINE_COMPILE_REQUIRED_EXT` to a success code, adding an alias for the original `VK_ERROR_PIPELINE_COMPILE_REQUIRED_EXT`. Also updated the xml to include these codes as return values.

`VK_EXT_pipeline_creation_feedback`

Name String

`VK_EXT_pipeline_creation_feedback`

Extension Type

Device extension

Registered Extension Number

193
Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Special Use
• Developer tools

Contact
• Jean-Francois Roy <jfroy>

Other Extension Metadata

Last Modified Date
2019-03-12

IP Status
No known IP claims.

Contributors
• Jean-Francois Roy, Google
• Hai Nguyen, Google
• Andrew Ellem, Google
• Bob Fraser, Google
• Sujeevan Rajayogam, Google
• Jan-Harald Fredriksen, ARM
• Jeff Leger, Qualcomm Technologies, Inc.
• Jeff Bolz, NVIDIA
• Daniel Koch, NVIDIA
• Neil Henning, AMD

Description
This extension adds a mechanism to provide feedback to an application about pipeline creation, with the specific goal of allowing a feedback loop between build systems and in-the-field application executions to ensure effective pipeline caches are shipped to customers.

New Structures
• VkPipelineCreationFeedbackEXT
• Extending VkGraphicsPipelineCreateInfo, VkComputePipelineCreateInfo, VkRayTracingPipelineCreateInfoNV, VkRayTracingPipelineCreateInfoKHR:
  • VkPipelineCreationFeedbackCreateInfoEXT
New Enums

• VkPipelineCreationFeedbackFlagBitsEXT

New Bitmasks

• VkPipelineCreationFeedbackFlagsEXT

New Enum Constants

• VK_EXT_PIPELINE_CREATION_FEEDBACK_EXTENSION_NAME
• VK_EXT_PIPELINE_CREATION_FEEDBACK_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PIPELINE_CREATION_FEEDBACK_CREATE_INFO_EXT

Version History

• Revision 1, 2019-03-12 (Jean-Francois Roy)
  ◦ Initial revision

VK_EXT_post_depth_coverage

Name String

VK_EXT_post_depth_coverage

Extension Type

Device extension

Registered Extension Number

156

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Daniel Koch dgkoch

Other Extension Metadata

Last Modified Date

2017-07-17

Interactions and External Dependencies

• This extension requires SPV_KHR_post_depth_coverage
This extension provides API support for GL_ARB_post_depth_coverage and GL_EXT_post_depth_coverage.

Contributors

• Jeff Bolz, NVIDIA

Description

This extension adds support for the following SPIR-V extension in Vulkan:

• SPV_KHR_post_depth_coverage

which allows the fragment shader to control whether values in the SampleMask built-in input variable reflect the coverage after early depth and stencil tests are applied.

This extension adds a new PostDepthCoverage execution mode under the SampleMaskPostDepthCoverage capability. When this mode is specified along with EarlyFragmentTests, the value of an input variable decorated with the SampleMask built-in reflects the coverage after the early fragment tests are applied. Otherwise, it reflects the coverage before the depth and stencil tests.

When using GLSL source-based shading languages, the post_depth_coverage layout qualifier from GL_ARB_post_depth_coverage or GL_EXT_post_depth_coverage maps to the PostDepthCoverage execution mode.

New Enum Constants

• VK_EXT_POST_DEPTH_COVERAGE_EXTENSION_NAME
  • VK_EXT_POST_DEPTH_COVERAGE_SPEC_VERSION

New SPIR-V Capabilities

• SampleMaskPostDepthCoverage

Version History

• Revision 1, 2017-07-17 (Daniel Koch)
  ◦ Internal revisions

VK_EXT_private_data

Name String
  VK_EXT_private_data

Extension Type
  Device extension

Registered Extension Number
  296
Description

The 'VK_EXT_private_data' extension is a device extension which enables attaching arbitrary payloads to Vulkan objects. It introduces the idea of private data slots as a means of storing a 64-bit unsigned integer of application defined data. Private data slots can be created or destroyed any time an associated device is available. Private data slots can be reserved at device creation time, and limiting use to the amount reserved will allow the extension to exhibit better performance characteristics.

New Object Types

- VkPrivateDataSlotEXT

New Commands

- vkCreatePrivateDataSlotEXT
- vkDestroyPrivateDataSlotEXT
- vkGetPrivateDataEXT
- vkSetPrivateDataEXT

New Structures

- VkPrivateDataSlotCreateInfoEXT
• Extending \texttt{VkDeviceCreateInfo}:
  ◦ \texttt{VkDevicePrivateDataCreateInfoEXT}

• Extending \texttt{VkPhysicalDeviceFeatures2, VkDeviceCreateInfo}:
  ◦ \texttt{VkPhysicalDevicePrivateDataFeaturesEXT}

\textbf{New Enums}

• \texttt{VkPrivateDataSlotCreateFlagBitsEXT}

\textbf{New Bitmasks}

• \texttt{VkPrivateDataSlotCreateFlagsEXT}

\textbf{New Enum Constants}

• \texttt{VK_EXT_PRIVATE_DATA_EXTENSION_NAME}
• \texttt{VK_EXT_PRIVATE_DATA_SPEC_VERSION}

• Extending \texttt{VkObjectType}:
  ◦ \texttt{VK_OBJECT_TYPE_PRIVATE_DATA_SLOT_EXT}

• Extending \texttt{VkStructureType}:
  ◦ \texttt{VK_STRUCTURE_TYPE_DEVICE_PRIVATE_DATA_CREATE_INFO_EXT}
  ◦ \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PRIVATE_DATA_FEATURES_EXT}
  ◦ \texttt{VK_STRUCTURE_TYPE_PRIVATE_DATA_SLOT_CREATE_INFO_EXT}

\textbf{Examples}

• In progress

\textbf{Version History}

• Revision 1, 2020-01-15 (Matthew Rusch)
  ◦ Initial draft

\textbf{VK_EXT_provoking_vertex}

\textbf{Name String}

\texttt{VK_EXT_provoking_vertex}

\textbf{Extension Type}

Device extension

\textbf{Registered Extension Number}

255
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Special Use

- OpenGL / ES support

Contact

- Jesse Hall @jessehall

Other Extension Metadata

Last Modified Date
2021-02-22

IP Status
No known IP claims.

Contributors

- Alexis Hétu, Google
- Bill Licea-Kane, Qualcomm
- Daniel Koch, Nvidia
- Jamie Madill, Google
- Jan-Harald Fredriksen, Arm
- Jason Ekstrand, Intel
- Jeff Bolz, Nvidia
- Jeff Leger, Qualcomm
- Jesse Hall, Google
- Jörg Wagner, Arm
- Matthew Netsch, Qualcomm
- Mike Blumenkrantz, Valve
- Piers Daniell, Nvidia
- Tobias Hector, AMD

Description

This extension allows changing the provoking vertex convention between Vulkan's default convention (first vertex) and OpenGL's convention (last vertex).

This extension is intended for use by API-translation layers that implement APIs like OpenGL on top...
of Vulkan, and need to match the source API's provoking vertex convention. Applications using
Vulkan directly should use Vulkan's default convention.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceProvokingVertexFeaturesEXT
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceProvokingVertexPropertiesEXT
- Extending VkPipelineRasterizationStateCreateInfo:
  - VkPipelineRasterizationProvokingVertexStateCreateInfoEXT

New Enums

- VkProvokingVertexModeEXT

New Enum Constants

- VK_EXT_PROVOKING_VERTEX_EXTENSION_NAME
- VK_EXT_PROVOKING_VERTEX_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROVOKING_VERTEX_FEATURES_EXT
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROVOKING_VERTEX_PROPERTIES_EXT
- VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_PROVOKING_VERTEX_STATE_CREATE_INFO_EXT

Issues

1) At what granularity should this state be set?

**Resolved**: At pipeline bind, with an optional per-renderpass restriction.

The most natural place to put this state is in the graphics pipeline object. Some implementa-
tions require it to be known when creating the pipeline, and pipeline state is convenient for
implementing OpenGL 3.2's glProvokingVertex, which can change the state between draw calls.
However, some implementations can only change it approximately renderpass granularity. To
accommodate both, provoking vertex will be pipeline state, but implementations can require that
only one mode is used within a renderpass instance; the renderpass's mode is chosen implicitly
when the first pipeline is bound.

2) Does the provoking vertex mode affect the order that vertices are written to transform feedback
buffers?

**Resolved**: Yes, to enable layered implementations of OpenGL and D3D.

All of OpenGL, OpenGL ES, and Direct3D 11 require that vertices are written to transform feedback
buffers such that flat-shaded attributes have the same value when drawing the contents of the
transform feedback buffer as they did in the original drawing when the transform feedback buffer
was written (assuming the provoking vertex mode hasn’t changed, in APIs that support more than
one mode).

**Version History**

- Revision 1(c), 2021-02-22 (Jesse Hall)
  - Added
    - `VkPhysicalDeviceProvokingVertexPropertiesEXT::transformFeedbackPreservesTriangleFanProvokingVertex` to accommodate implementations that can’t change the transform feedback vertex order for triangle fans.

- Revision 1(b), 2020-06-14 (Jesse Hall)
  - Added
    - `VkPhysicalDeviceProvokingVertexFeaturesEXT::transformFeedbackPreservesProvokingVertex` and required that transform feedback write vertices so as to preserve the provoking vertex of each primitive.

- Revision 1(a), 2019-10-23 (Jesse Hall)
  - Initial draft, based on a proposal by Alexis Hétu

**VK_EXT_queue_family_foreign**

**Name String**

`VK_EXT_queue_family_foreign`

**Extension Type**

Device extension

**Registered Extension Number**

127

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_KHR_external_memory`

**Contact**

- Chad Versace [chadversary](mailto:chadversary)

**Other Extension Metadata**

**Last Modified Date**

2017-11-01
IP Status
No known IP claims.

Contributors
• Chad Versace, Google
• James Jones, NVIDIA
• Jason Ekstrand, Intel
• Jesse Hall, Google
• Daniel Rakos, AMD
• Ray Smith, ARM

Description
This extension defines a special queue family, VK_QUEUE_FAMILY_FOREIGN_EXT, which can be used to transfer ownership of resources backed by external memory to foreign, external queues. This is similar to VK_QUEUE_FAMILY_EXTERNAL_KHR, defined in VK_KHR_external_memory. The key differences between the two are:

• The queues represented by VK_QUEUE_FAMILY_EXTERNAL_KHR must share the same physical device and the same driver version as the current VkInstance. VK_QUEUE_FAMILY_FOREIGN_EXT has no such restrictions. It can represent devices and drivers from other vendors, and can even represent non-Vulkan-capable devices.

• All resources backed by external memory support VK_QUEUE_FAMILY_EXTERNAL_KHR. Support for VK_QUEUE_FAMILY_FOREIGN_EXT is more restrictive.

• Applications should expect transitions to/from VK_QUEUE_FAMILY_FOREIGN_EXT to be more expensive than transitions to/from VK_QUEUE_FAMILY_EXTERNAL_KHR.

New Enum Constants
• VK_EXT_QUEUE_FAMILY_FOREIGN_EXTENSION_NAME
• VK_EXT_QUEUE_FAMILY_FOREIGN_SPEC_VERSION
• VK_QUEUE_FAMILY_FOREIGN_EXT

Version History
• Revision 1, 2017-11-01 (Chad Versace)
  ◦ Squashed internal revisions

VK_EXT_robustness2

Name String
VK_EXT_robustness2
Extension Type
  Device extension

Registered Extension Number
  287

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0

Contact
  • Liam Middlebrook liam-middlebrook

Other Extension Metadata

Last Modified Date
  2020-01-29

IP Status
  No known IP claims.

Contributors
  • Liam Middlebrook, NVIDIA
  • Jeff Bolz, NVIDIA

Description

This extension adds stricter requirements for how out of bounds reads and writes are handled. Most accesses must be tightly bounds-checked, out of bounds writes must be discarded, out of bound reads must return zero. Rather than allowing multiple possible (0,0,0,x) vectors, the out of bounds values are treated as zero, and then missing components are inserted based on the format as described in Conversion to RGBA and vertex input attribute extraction.

These additional requirements may be expensive on some implementations, and should only be enabled when truly necessary.

This extension also adds support for “null descriptors”, where VK_NULL_HANDLE can be used instead of a valid handle. Accesses to null descriptors have well-defined behavior, and don’t rely on robustness.

New Structures

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceRobustness2FeaturesEXT
• Extending VkPhysicalDeviceProperties2:
New Enum Constants

- `VK_EXT_ROBUSTNESS_2_EXTENSION_NAME`
- `VK_EXT_ROBUSTNESS_2_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ROBUSTNESS_2_PROPERTIES_EXT`

Issues

1. **Why do** `VkPhysicalDeviceRobustness2PropertiesEXT::robustUniformBufferAccessSizeAlignment` and `VkPhysicalDeviceRobustness2PropertiesEXT::robustStorageBufferAccessSizeAlignment` **exist?**

**Resolved:** Some implementations can't efficiently tightly bounds-check all buffer accesses. Rather, the size of the bound range is padded to some power of two multiple, up to 256 bytes for uniform buffers and up to 4 bytes for storage buffers, and that padded size is bounds-checked. This is sufficient to implement D3D-like behavior, because D3D only allows binding whole uniform buffers or ranges that are a multiple of 256 bytes, and D3D raw and structured buffers only support 32-bit accesses.

Examples

None.

Version History

- Revision 1, 2019-11-01 (Jeff Bolz, Liam Middlebrook)
  - Initial draft

**VK_EXT_sample_locations**

**Name String**

`VK_EXT_sample_locations`

**Extension Type**

Device extension

**Registered Extension Number**

144

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
Description

This extension allows an application to modify the locations of samples within a pixel used in rasterization. Additionally, it allows applications to specify different sample locations for each pixel in a group of adjacent pixels, which can increase antialiasing quality (particularly if a custom resolve shader is used that takes advantage of these different locations).

It is common for implementations to optimize the storage of depth values by storing values that can be used to reconstruct depth at each sample location, rather than storing separate depth values for each sample. For example, the depth values from a single triangle may be represented using plane equations. When the depth value for a sample is needed, it is automatically evaluated at the sample location. Modifying the sample locations causes the reconstruction to no longer evaluate the same depth values as when the samples were originally generated, thus the depth aspect of a depth/stencil attachment must be cleared before rendering to it using different sample locations.

Some implementations may need to evaluate depth image values while performing image layout transitions. To accommodate this, instances of the VkSampleLocationsInfoEXT structure can be specified for each situation where an explicit or automatic layout transition has to take place. VkSampleLocationsInfoEXT can be chained from VkImageMemoryBarrier structures to provide sample locations for layout transitions performed by vkCmdWaitEvents and vkCmdPipelineBarrier calls, and VkRenderPassSampleLocationsBeginInfoEXT can be chained from VkRenderPassBeginInfo to provide sample locations for layout transitions performed implicitly by a render pass instance.

New Commands

- vkCmdSetSampleLocationsEXT
• vkGetPhysicalDeviceMultisamplePropertiesEXT

New Structures

• VkAttachmentSampleLocationsEXT
• VkMultisamplePropertiesEXT
• VkSampleLocationEXT
• VkSubpassSampleLocationsEXT
• Extending VkImageMemoryBarrier, VkImageMemoryBarrier2KHR:
  ◦ VkSampleLocationsInfoEXT
• Extending VkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceSampleLocationsPropertiesEXT
• Extending VkPipelineMultisampleStateCreateInfo:
  ◦ VkPipelineSampleLocationsStateCreateInfoEXT
• Extending VkRenderPassBeginInfo:
  ◦ VkRenderPassSampleLocationsBeginInfoEXT

New Enum Constants

• VK_EXT_SAMPLE_LOCATIONS_EXTENSION_NAME
• VK_EXT_SAMPLE_LOCATIONS_SPEC_VERSION
• Extending VkDynamicState:
  ◦ VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT
• Extending VkImageCreateFlagBits:
  ◦ VK_IMAGE_CREATE_SAMPLE_LOCATIONS_COMPATIBLE_DEPTH_BIT_EXT
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_MULTISAMPLE_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLE_LOCATIONS_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_PIPELINE_SAMPLE_LOCATIONS_STATE_CREATE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_RENDER_PASS_SAMPLE_LOCATIONS_BEGIN_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_SAMPLE_LOCATIONS_INFO_EXT

Version History

• Revision 1, 2017-08-02 (Daniel Rakos)
  ◦ Internal revisions
VK_EXT_sampler_filter_minmax

Name String
VK_EXT_sampler_filter_minmax

Extension Type
Device extension

Registered Extension Number
131

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Deprecation state
• Promoted to Vulkan 1.2

Contact
• Jeff Bolz jeffbolznv

Other Extension Metadata

Last Modified Date
2017-05-19

Interactions and External Dependencies
• Promoted to Vulkan 1.2 Core

IP Status
No known IP claims.

Contributors
• Jeff Bolz, NVIDIA
• Piers Daniell, NVIDIA

Description
In unextended Vulkan, minification and magnification filters such as LINEAR allow sampled image lookups to return a filtered texel value produced by computing a weighted average of a collection of texels in the neighborhood of the texture coordinate provided.

This extension provides a new sampler parameter which allows applications to produce a filtered texel value by computing a component-wise minimum (MIN) or maximum (MAX) of the texels that would normally be averaged. The reduction mode is orthogonal to the minification and
magnification filter parameters. The filter parameters are used to identify the set of texels used to produce a final filtered value; the reduction mode identifies how these texels are combined.

**Promotion to Vulkan 1.2**

All functionality in this extension is included in core Vulkan 1.2, with the EXT suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

**New Structures**

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceSamplerFilterMinmaxPropertiesEXT`

- Extending `VkSamplerCreateInfo`:
  - `VkSamplerReductionModeCreateInfoEXT`

**New Enums**

- `VkSamplerReductionModeEXT`

**New Enum Constants**

- `VK_EXT_SAMPLER_FILTER_MINMAX_EXTENSION_NAME`
- `VK_EXT_SAMPLER_FILTER_MINMAX_SPEC_VERSION`

**Extending `VkFormatFeatureFlagBits`**:

- `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_MINMAX_BIT_EXT`

**Extending `VkSamplerReductionMode`**:

- `VK_SAMPLER_REDUCTION_MODE_MAX_EXT`
- `VK_SAMPLER_REDUCTION_MODE_MIN_EXT`
- `VK_SAMPLER_REDUCTION_MODE_WEIGHTED_AVERAGE_EXT`

**Extending `VkStructureType`**:

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_FILTER_MINMAX_PROPERTIES_EXT`
- `VK_STRUCTURE_TYPE_SAMPLER_REDUCTION_MODE_CREATE_INFO_EXT`

**Version History**

- Revision 2, 2017-05-19 (Piers Daniell)
  - Renamed to EXT

- Revision 1, 2017-03-25 (Jeff Bolz)
  - Internal revisions
**VK_EXT_scalar_block_layout**

**Name String**

VK_EXT_scalar_block_layout

**Extension Type**

Device extension

**Registered Extension Number**

222

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Deprecation state**

- Promoted to Vulkan 1.2

**Contact**

- Tobias Hector (tobski)

**Other Extension Metadata**

**Last Modified Date**

2018-11-14

**Interactions and External Dependencies**

- Promoted to Vulkan 1.2 Core

**Contributors**

- Jeff Bolz
- Jan-Harald Fredriksen
- Graeme Leese
- Jason Ekstrand
- John Kessenich

**Description**

This extension enables C-like structure layout for SPIR-V blocks. It modifies the alignment rules for uniform buffers, storage buffers and push constants, allowing non-scalar types to be aligned solely based on the size of their components, without additional requirements.
Promotion to Vulkan 1.2

Functionality in this extension is included in core Vulkan 1.2, with the EXT suffix omitted. However, if Vulkan 1.2 is supported and this extension is not, the `scalarBlockLayout` capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceScalarBlockLayoutFeaturesEXT`

New Enum Constants

- `VK_EXT_SCALAR_BLOCK_LAYOUT_EXTENSION_NAME`
- `VK_EXT_SCALAR_BLOCK_LAYOUT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SCALAR_BLOCK_LAYOUT_FEATURES_EXT`

Version History

- Revision 1, 2018-11-14 (Tobias Hector)
  - Initial draft

`VK_EXT_separate_stencil_usage`

Name String

- `VK_EXT_separate_stencil_usage`

Extension Type

- Device extension

Registered Extension Number

- 247

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- *Promoted* to Vulkan 1.2

Contact

- Daniel Rakos [drakos-amd](mailto:drakos-amd)
Other Extension Metadata

Last Modified Date
2018-11-08

Interactions and External Dependencies
- Promoted to Vulkan 1.2 Core

IP Status
No known IP claims.

Contributors
- Daniel Rakos, AMD
- Jordan Logan, AMD

Description
This extension allows specifying separate usage flags for the stencil aspect of images with a depth-stencil format at image creation time.

Promotion to Vulkan 1.2
All functionality in this extension is included in core Vulkan 1.2, with the EXT suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures
- Extending `VkImageCreateInfo`, `VkPhysicalDeviceImageFormatInfo2`:
  - `VkImageStencilUsageCreateInfoEXT`

New Enum Constants
- `VK_EXT_SEPARATE_STENCIL_USAGE_EXTENSION_NAME`
- `VK_EXT_SEPARATE_STENCIL_USAGE_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_IMAGE_STENCIL_USAGE_CREATE_INFO_EXT`

Version History
- Revision 1, 2018-11-08 (Daniel Rakos)
  - Internal revisions.

VK_EXT_shader_atomic_float

Name String
- `VK_EXT_shader_atomic_float`
Extension Type
Device extension

Registered Extension Number
261

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact
• Vikram Kushwaha vkushwaha-nv

Other Extension Metadata

Last Modified Date
2020-07-15

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_EXT_shader_atomic_float_add
• This extension provides API support for GL_EXT_shader_atomic_float

Contributors
• Vikram Kushwaha, NVIDIA
• Jeff Bolz, NVIDIA

Description
This extension allows a shader to contain floating-point atomic operations on buffer, workgroup, and image memory. It also advertises the SPIR-V AtomicFloat32AddEXT and AtomicFloat64AddEXT capabilities that allows atomic addition on floating-points numbers. The supported operations include OpAtomicFAddEXT, OpAtomicExchange, OpAtomicLoad and OpAtomicStore.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  – VkPhysicalDeviceShaderAtomicFloatFeaturesEXT
New Enum Constants

- VK_EXT_SHADER_ATOMIC_FLOAT_EXTENSION_NAME
- VK_EXT_SHADER_ATOMIC_FLOAT_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_ATOMIC_FLOAT_FEATURES_EXT

New SPIR-V Capabilities

- AtomicFloat32AddEXT
- AtomicFloat64AddEXT

Version History

- Revision 1, 2020-07-15 (Vikram Kushwaha)
  - Internal revisions

VK_EXT_shader_demote_to_helper_invocation

Name String
- VK_EXT_shader_demote_to_helper_invocation

Extension Type
- Device extension

Registered Extension Number
- 277

Revision
- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Jeff Bolz @jeffbolznv

Other Extension Metadata

Last Modified Date
- 2019-06-01

IP Status
- No known IP claims.
Interactions and External Dependencies

- This extension requires `SPV_EXT_demote_to_helper_invocation`

Contributors

- Jeff Bolz, NVIDIA

Description

This extension adds Vulkan support for the `SPV_EXT_demote_to_helper_invocation` SPIR-V extension. That SPIR-V extension provides a new instruction `OpDemoteToHelperInvocationEXT` allowing shaders to “demote” a fragment shader invocation to behave like a helper invocation for its duration. The demoted invocation will have no further side effects and will not output to the framebuffer, but remains active and can participate in computing derivatives and in group operations. This is a better match for the “discard” instruction in HLSL.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceShaderDemoteToHelperInvocationFeaturesEXT`

New Enum Constants

- `VK_EXT_SHADER_DEMOTE_TO_HELPER_INVOCATION_EXTENSION_NAME`
- `VK_EXT_SHADER_DEMOTE_TO_HELPER_INVOCATION_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_DEMOTE_TO_HELPER_INVOCATION_FEATURES_EXT`

New SPIR-V Capability

- `DemoteToHelperInvocationEXT`

Version History

- Revision 1, 2019-06-01 (Jeff Bolz)
  - Initial draft

`VK_EXT_shader_image_atomic_int64`

Name String

`VK_EXT_shader_image_atomic_int64`

Extension Type

- Device extension

Registered Extension Number

- 235
Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Tobias Hector @tobski

Other Extension Metadata

Last Modified Date

2020-07-14

IP Status

No known IP claims.

Contributors

• Matthaeus Chajdas, AMD
• Graham Wihlidal, Epic Games
• Tobias Hector, AMD
• Jeff Bolz, Nvidia
• Jason Ekstrand, Intel

Interactions and External Dependencies

• This extension requires the SPV_EXT_shader_image_int64 SPIR-V extension.
• This extension requires the GLSL_EXT_shader_image_int64 extension for GLSL source languages.

Description

This extension extends existing 64-bit integer atomic support to enable these operations on images as well.

When working with large 2- or 3-dimensional data sets (e.g. rasterization or screen-space effects), image accesses are generally more efficient than equivalent buffer accesses. This extension allows applications relying on 64-bit integer atomics in this manner to quickly improve performance with only relatively minor code changes.

64-bit integer atomic support is guaranteed for optimally tiled images with the VK_FORMAT_R64_UINT and VK_FORMAT_R64_SINT formats.

New Structures

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
New Enum Constants

- VK_EXT_SHADER_IMAGE_ATOMIC_INT64_EXTENSION_NAME
- VK_EXT_SHADER_IMAGE_ATOMIC_INT64_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_ATOMIC_INT64_FEATURES_EXT

Version History

- Revision 1, 2020-07-14 (Tobias Hector)
  - Initial draft

VK_EXT_shader_stencil_export

Name String

VK_EXT_shader_stencil_export

Extension Type

Device extension

Registered Extension Number

141

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Dominik Witczak dominikwitczakamd

Other Extension Metadata

Last Modified Date

2017-07-19

IP Status

No known IP claims.

Interactions and External Dependencies

- This extension requires SPV_EXT_shader_stencil_export
- This extension provides API support for GL_ARB_shader_stencil_export
Contributors

- Dominik Witczak, AMD
- Daniel Rakos, AMD
- Rex Xu, AMD

Description

This extension adds support for the SPIR-V extension SPV_EXT_shader_stencil_export, providing a mechanism whereby a shader may generate the stencil reference value per invocation. When stencil testing is enabled, this allows the test to be performed against the value generated in the shader.

New Enum Constants

- VK_EXT_SHADER_STENCIL_EXPORT_EXTENSION_NAME
- VK_EXT_SHADER_STENCIL_EXPORT_SPEC_VERSION

Version History

- Revision 1, 2017-07-19 (Dominik Witczak)
  - Initial draft

VK_EXT_shader_subgroup_ballot

Name String

VK_EXT_shader_subgroup_ballot

Extension Type

Device extension

Registered Extension Number

65

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Deprecated by Vulkan 1.2

Contact

- Daniel Koch dgkoch
Other Extension Metadata

Last Modified Date
2016-11-28

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_KHR_shader_ballot
• This extension provides API support for GL_ARB_shader_ballot

Contributors
• Jeff Bolz, NVIDIA
• Neil Henning, Codeplay
• Daniel Koch, NVIDIA Corporation

Description
This extension adds support for the following SPIR-V extension in Vulkan:
• SPV_KHR_shader_ballot

This extension provides the ability for a group of invocations, which execute in parallel, to do limited forms of cross-invocation communication via a group broadcast of a invocation value, or broadcast of a bitarray representing a predicate value from each invocation in the group.

This extension provides access to a number of additional built-in shader variables in Vulkan:
• SubgroupEqMaskKHR, which contains the subgroup mask of the current subgroup invocation,
• SubgroupGeMaskKHR, which contains the subgroup mask of the invocations greater than or equal to the current invocation,
• SubgroupGtMaskKHR, which contains the subgroup mask of the invocations greater than the current invocation,
• SubgroupLeMaskKHR, which contains the subgroup mask of the invocations less than or equal to the current invocation,
• SubgroupLtMaskKHR, which contains the subgroup mask of the invocations less than the current invocation,
• SubgroupLocalInvocationId, which contains the index of an invocation within a subgroup, and
• SubgroupSize, which contains the maximum number of invocations in a subgroup.

Additionally, this extension provides access to the new SPIR-V instructions:
• OpSubgroupBallotKHR,
• OpSubgroupFirstInvocationKHR, and
When using GLSL source-based shader languages, the following variables and shader functions from GL_ARB_shader_ballot can map to these SPIR-V built-in decorations and instructions:

- `in uint64_t gl_SubGroupEqMaskARB; → SubgroupEqMaskKHR,`
- `in uint64_t gl_SubGroupGeMaskARB; → SubgroupGeMaskKHR,`
- `in uint64_t gl_SubGroupGtMaskARB; → SubgroupGtMaskKHR,`
- `in uint64_t gl_SubGroupLeMaskARB; → SubgroupLeMaskKHR,`
- `in uint64_t gl_SubGroupLtMaskARB; → SubgroupLtMaskKHR,`
- `in uint gl_SubGroupInvocationARB; → SubgroupLocalInvocationId,`
- `uniform uint gl_SubGroupSizeARB; → SubgroupSize,`
- `ballotARB() → OpSubgroupBallotKHR,`
- `readFirstInvocationARB() → OpSubgroupFirstInvocationKHR, and`
- `readInvocationARB() → OpSubgroupReadInvocationKHR.

**Deprecated by Vulkan 1.2**

Most of the functionality in this extension is superseded by the core Vulkan 1.1 subgroup operations. However, Vulkan 1.1 required the `OpGroupNonUniformBroadcast “Id”` to be constant. This restriction was removed in Vulkan 1.2 with the addition of the `subgroupBroadcastDynamicId` feature.

**New Enum Constants**

- `VK_EXT_SHADER_SUBGROUP_BALLOT_EXTENSION_NAME`
- `VK_EXT_SHADER_SUBGROUP_BALLOT_SPEC_VERSION`

**New Built-In Variables**

- `SubgroupEqMaskKHR`
- `SubgroupGeMaskKHR`
- `SubgroupGtMaskKHR`
- `SubgroupLeMaskKHR`
- `SubgroupLtMaskKHR`
- `SubgroupLocalInvocationId`
- `SubgroupSize`

**New SPIR-V Capabilities**

- `SubgroupBallotKHR`
Version History

- Revision 1, 2016-11-28 (Daniel Koch)
  - Initial draft

**VK_EXT_shader_viewport_index_layer**

Name String

VK_EXT_shader_viewport_index_layer

Extension Type

Device extension

Registered Extension Number

163

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Promoted to Vulkan 1.2

Contact

- Daniel Koch 🍀dgkoch

Other Extension Metadata

Last Modified Date

2017-08-08

Interactions and External Dependencies

- Promoted to Vulkan 1.2 Core
- This extension requires SPV_EXT_shader_viewport_index_layer
- This extension provides API support for GL_ARB_shader_viewport_layer_array, GL_AMD_vertex_shader_layer, GL_AMD_vertex_shader_viewport_index, and GL_NV_viewport_array2
- This extension requires the multiViewport feature.
- This extension interacts with the tessellationShader feature.

Contributors

- Piers Daniell, NVIDIA
- Jeff Bolz, NVIDIA
- Jan-Harald Fredriksen, ARM
Description

This extension adds support for the `ShaderViewportIndexLayerEXT` capability from the SPV_EXT_shader_viewport_index_layer extension in Vulkan.

This extension allows variables decorated with the `Layer` and `ViewportIndex` built-ins to be exported from vertex or tessellation shaders, using the `ShaderViewportIndexLayerEXT` capability.

When using GLSL source-based shading languages, the `gl_ViewportIndex` and `gl_Layer` built-in variables map to the SPIR-V `ViewportIndex` and `Layer` built-in decorations, respectively. Behaviour of these variables is extended as described in the GL_ARB_shader_viewport_layer_array (or the precursor GL_AMD_vertex_shader_layer, GL_AMD_vertex_shader_viewport_index, and GL_NV_viewport_array2 extensions).

Note

The `ShaderViewportIndexLayerEXT` capability is equivalent to the `ShaderViewportIndexLayerNV` capability added by VK_NV_viewport_array2.

Promotion to Vulkan 1.2

All functionality in this extension is included in core Vulkan 1.2.

The single `ShaderViewportIndexLayerEXT` capability from the SPV_EXT_shader_viewport_index_layer extension is replaced by the `ShaderViewportIndex` and `ShaderLayer` capabilities from SPIR-V 1.5 which are enabled by the `shaderOutputViewportIndex` and `shaderOutputLayer` features, respectively. Additionally, if Vulkan 1.2 is supported but this extension is not, these capabilities are optional.

Enabling both features is equivalent to enabling the VK_EXT_shader_viewport_index_layer extension.

New Enum Constants

- VK_EXT_SHADER_VIEWPORT_INDEX_LAYER_EXTENSION_NAME
- VK_EXT_SHADER_VIEWPORT_INDEX_LAYER_SPEC_VERSION

New or Modified Built-In Variables

- (modified) `Layer`
- (modified) `ViewportIndex`

New SPIR-V Capabilities

- `ShaderViewportIndexLayerEXT`
Version History

- Revision 1, 2017-08-08 (Daniel Koch)
  - Internal drafts

VK_EXT_subgroup_size_control

Name String

VK_EXT_subgroup_size_control

Extension Type

Device extension

Registered Extension Number

226

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.1

Contact

- Neil Henning sheredom

Other Extension Metadata

Last Modified Date

2019-03-05

Contributors

- Jeff Bolz, NVIDIA
- Jason Ekstrand, Intel
- Sławek Grajewski, Intel
- Jesse Hall, Google
- Neil Henning, AMD
- Daniel Koch, NVIDIA
- Jeff Leger, Qualcomm
- Graeme Leese, Broadcom
- Allan MacKinnon, Google
- Mariusz Merecki, Intel
- Graham Wihlidal, Electronic Arts
Description

This extension enables an implementation to control the subgroup size by allowing a varying subgroup size and also specifying a required subgroup size.

It extends the subgroup support in Vulkan 1.1 to allow an implementation to expose a varying subgroup size. Previously Vulkan exposed a single subgroup size per physical device, with the expectation that implementations will behave as if all subgroups have the same size. Some implementations may dispatch shaders with a varying subgroup size for different subgroups. As a result they could implicitly split a large subgroup into smaller subgroups or represent a small subgroup as a larger subgroup, some of whose invocations were inactive on launch.

To aid developers in understanding the performance characteristics of their programs, this extension exposes a minimum and maximum subgroup size that a physical device supports and a pipeline create flag to enable that pipeline to vary its subgroup size. If enabled, any SubgroupSize decorated variables in the SPIR-V shader modules provided to pipeline creation may vary between the minimum and maximum subgroup sizes.

An implementation is also optionally allowed to support specifying a required subgroup size for a given pipeline stage. Implementations advertise which stages support a required subgroup size, and any pipeline of a supported stage can be passed a VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT structure to set the subgroup size for that shader stage of the pipeline. For compute shaders, this requires the developer to query the maxComputeWorkgroupSubgroups and ensure that:

\[ s = \text{WorkGroupSize.x} \times \text{WorkGroupSize.y} \times \text{WorkGroupSize.z} \leq \text{SubgroupSize} \times \text{maxComputeWorkgroupSubgroups} \]

Developers can also specify a new pipeline shader stage create flag that requires the implementation to have fully populated subgroups within local workgroups. This requires the workgroup size in the X dimension to be a multiple of the subgroup size.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceSubgroupSizeControlFeaturesEXT

- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceSubgroupSizeControlPropertiesEXT

- Extending VkPipelineShaderStageCreateInfo:
  - VkPipelineShaderStageRequiredSubgroupSizeCreateInfoEXT

New Enum Constants

- VK_EXT_SUBGROUP_SIZE_CONTROL_EXTENSION_NAME
- VK_EXT_SUBGROUP_SIZE_CONTROL_SPEC_VERSION
- Extending VkPipelineShaderStageCreateFlagBits:
  - VK_PIPELINE_SHADER_STAGE_CREATE_ALLOW_VARYING_SUBGROUP_SIZE_BIT_EXT
• Extending \texttt{VkStructureType}:
  
  \begin{itemize}
  \item \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_FEATURES_EXT}
  \item \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SUBGROUP_SIZE_CONTROL_PROPERTIES_EXT}
  \item \texttt{VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_REQUIRED_SUBGROUP_SIZE_CREATE_INFO_EXT}
  \end{itemize}

**Version History**

- Revision 1, 2019-03-05 (Neil Henning)
  - Initial draft
- Revision 2, 2019-07-26 (Jason Ekstrand)
  - Add the missing \texttt{VkPhysicalDeviceSubgroupSizeControlFeaturesEXT} for querying subgroup size control features.

**VK_EXT_swapchain_colorspace**

**Name String**

\texttt{VK_EXT_swapchain_colorspace}

**Extension Type**

- Instance extension

**Registered Extension Number**

- 105

**Revision**

- 4

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires \texttt{VK_KHR_surface}

**Contact**

- Courtney Goeltzenleuchter [courtney-g](mailto:courtney-g)

**Other Extension Metadata**

**Last Modified Date**

- 2019-04-26

**IP Status**

- No known IP claims.

**Contributors**

- Courtney Goeltzenleuchter, Google
Description

To be done.

New Enum Constants

- VK_EXT_SWAPCHAIN_COLOR_SPACE_EXTENSION_NAME
- VK_EXT_SWAPCHAIN_COLOR_SPACE_SPEC_VERSION

Extending VkColorSpaceKHR:

- VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT
- VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT
- VK_COLOR_SPACE_BT2020_LINEAR_EXT
- VK_COLOR_SPACE_BT709_LINEAR_EXT
- VK_COLOR_SPACE_BT709_NONLINEAR_EXT
- VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT
- VK_COLOR_SPACE_DISPLAY_P3_LINEAR_EXT
- VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT
- VK_COLOR_SPACE_DOLBYVISION_EXT
- VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT
- VK_COLOR_SPACE_EXTENDED_SRGB_NONLINEAR_EXT
- VK_COLOR_SPACE_HDR10_HLG_EXT
- VK_COLOR_SPACE_HDR10_ST2084_EXT
- VK_COLOR_SPACE_PASS_THROUGH_EXT

Issues

1) Does the spec need to specify which kinds of image formats support the color spaces?

**RESOLVED:** Pixel format is independent of color space (though some color spaces really want / need floating point color components to be useful). Therefore, do not plan on documenting what formats support which colorspace. An application can call vkGetPhysicalDeviceSurfaceFormatsKHR to query what a particular implementation supports.

2) How does application determine if HW supports appropriate transfer function for a colorspace?

**RESOLVED:** Extension indicates that implementation must not do the OETF encoding if it is not sRGB. That responsibility falls to the application shaders. Any other native OETF / EOTF functions supported by an implementation can be described by separate extension.

Version History

- Revision 1, 2016-12-27 (Courtney Goeltzenleuchter)
Initial version

- Revision 2, 2017-01-19 (Courtney Goeltzenleuchter)
  - Add pass through and multiple options for BT2020.
  - Clean up some issues with equations not displaying properly.

- Revision 3, 2017-06-23 (Courtney Goeltzenleuchter)
  - Add extended sRGB non-linear enum.

- Revision 4, 2019-04-26 (Graeme Leese)
  - Clarify colorspace transfer function usage.
  - Refer to normative definitions in the Data Format Specification.
  - Clarify DCI-P3 and Display P3 usage.

**VK_EXT_texel_buffer_alignment**

**Name String**

VK_EXT_texel_buffer_alignment

**Extension Type**

Device extension

**Registered Extension Number**

282

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

**Other Extension Metadata**

**Last Modified Date**

2019-06-06

**IP Status**

No known IP claims.

**Interactions and External Dependencies**

**Contributors**

- Jeff Bolz, NVIDIA
Description

This extension adds more expressive alignment requirements for uniform and storage texel buffers. Some implementations have single texel alignment requirements that can’t be expressed via `VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment`.

New Structures

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceTexelBufferAlignmentFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceTexelBufferAlignmentPropertiesEXT`

New Enum Constants

- `VK_EXT_TEXEL_BUFFER_ALIGNMENT_EXTENSION_NAME`
- `VK_EXT_TEXEL_BUFFER_ALIGNMENT_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_FEATURES_EXT`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXEL_BUFFER_ALIGNMENT_PROPERTIES_EXT`

Version History

- Revision 1, 2019-06-06 (Jeff Bolz)
  - Initial draft

VK_EXT_texture_compression_astc_hdr

Name String

`VK_EXT_texture_compression_astc_hdr`

Extension Type

Device extension

Registered Extension Number

67

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`
Contact

- Jan-Harald Fredriksen janharaldfredriksen-arm

Other Extension Metadata

Last Modified Date
2019-05-28

IP Status
No known issues.

Contributors

- Jan-Harald Fredriksen, Arm

Description

This extension adds support for textures compressed using the Adaptive Scalable Texture Compression (ASTC) High Dynamic Range (HDR) profile.

When this extension is enabled, the HDR profile is supported for all ASTC formats listed in ASTC Compressed Image Formats.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceTextureCompressionASTCHDRFeaturesEXT

New Enum Constants

- VK_EXT_TEXTURE_COMPRESSION_ASTC_HDR_EXTENSION_NAME
- VK_EXT_TEXTURE_COMPRESSION_ASTC_HDR_SPEC_VERSION

- Extending VkFormat:
  - VK_FORMAT_ASTC_10x10_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_10x5_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_10x6_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_10x8_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_12x10_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_12x12_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_4x4_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_5x4_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_5x5_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_6x5_SFLOAT_BLOCK_EXT
  - VK_FORMAT_ASTC_6x6_SFLOAT_BLOCK_EXT
• **Extending **VkStructureType:

  • `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TEXTURE_COMPRESSION_ASTC_HDR_FEATURES_EXT`

**Issues**

1) Should we add a feature or limit for this functionality?

Yes. It is consistent with the ASTC LDR support to add a feature like textureCompressionASTC_HDR. The feature is strictly speaking redundant as long as this is just an extension; it would be sufficient to just enable the extension. But adding the feature is more forward-looking if wanted to make this an optional core feature in the future.

2) Should we introduce new format enums for HDR?

Yes. Vulkan 1.0 describes the ASTC format enums as UNORM, e.g. `VK_FORMAT_ASTC_4x4_UNORM_BLOCK`, so it’s confusing to make these contain HDR data. Note that the OpenGL (ES) extensions did not make this distinction because a single ASTC HDR texture may contain both unorm and float blocks. Implementations may not be able to distinguish between LDR and HDR ASTC textures internally and just treat them as the same format, i.e. if this extension is supported then sampling from a `VK_FORMAT_ASTC_4x4_UNORM_BLOCK` image format may return HDR results. Applications can get predictable results by using the appropriate image format.

**Version History**

• Revision 1, 2019-05-28 (Jan-Harald Fredriksen)
  • Initial version

**VK_EXT_tooling_info**

**Name String**

  `VK_EXT_tooling_info`

**Extension Type**

  Device extension

**Registered Extension Number**

  246

**Revision**

  1

**Extension and Version Dependencies**

  • Requires Vulkan 1.0
Description

When an error occurs during application development, a common question is "What tools are actually running right now?" This extension adds the ability to query that information directly from the Vulkan implementation.

Outdated versions of one tool might not play nicely with another, or perhaps a tool is not actually running when it should have been. Trying to figure that out can cause headaches as it is necessary to consult each known tool to figure out what is going on—in some cases the tool might not even be known.

Typically, the expectation is that developers will simply print out this information for visual inspection when an issue occurs, however a small amount of semantic information about what the tool is doing is provided to help identify it programmatically. For example, if the advertised limits or features of an implementation are unexpected, is there a tool active which modifies these limits? Or if an application is providing debug markers, but the implementation is not actually doing anything with that information, this can quickly point that out.

New Commands

• vkGetPhysicalDeviceToolPropertiesEXT

New Structures

• VkPhysicalDeviceToolPropertiesEXT

New Enums

• VkToolPurposeFlagBitsEXT

New Bitmasks

• VkToolPurposeFlagsEXT
New Enum Constants

- `VK_EXT_TOOLING_INFO_EXTENSION_NAME`
- `VK_EXT_TOOLING_INFO_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TOOL_PROPERTIES_EXT`

If `VK_EXT_debug_marker` is supported:

- Extending `VkToolPurposeFlagBitsEXT`:
  - `VK_TOOL_PURPOSE_DEBUG_MARKERS_BIT_EXT`

If `VK_EXT_debug_report` is supported:

- Extending `VkToolPurposeFlagBitsEXT`:
  - `VK_TOOL_PURPOSE_DEBUG_REPORTING_BIT_EXT`

If `VK_EXT_debug_utils` is supported:

- Extending `VkToolPurposeFlagBitsEXT`:
  - `VK_TOOL_PURPOSE_DEBUG_MARKERS_BIT_EXT`
  - `VK_TOOL_PURPOSE_DEBUG_REPORTING_BIT_EXT`

Examples
uint32_t num_tools;
VkPhysicalDeviceToolPropertiesEXT *pToolProperties;
vkGetPhysicalDeviceToolPropertiesEXT(physicalDevice, &num_tools, NULL);

pToolProperties = (VkPhysicalDeviceToolPropertiesEXT*)malloc(sizeof(VkPhysicalDeviceToolPropertiesEXT) * num_tools);

vkGetPhysicalDeviceToolPropertiesEXT(physicalDevice, &num_tools, pToolProperties);

for (int i = 0; i < num_tools; ++i) {
    printf("%s:\n", pToolProperties[i].name);
    printf("Version:\n");
    printf("%s:\n", pToolProperties[i].version);
    printf("Description:\n");
    printf("\t%s\n", pToolProperties[i].description);
    printf("Purposes:\n");
    printf("\t%s\n", VkToolPurposeFlagBitsEXT_to_string(pToolProperties[i].purposes));
    if (strnlen_s(pToolProperties[i].layer,VK_MAX_EXTENSION_NAME_SIZE) > 0) {
        printf("Corresponding Layer:\n");
        printf("\t%s\n", pToolProperties[i].layer);
    }
}

**Issues**

1) Why is this information separate from the layer mechanism?

Some tooing may be built into a driver, or be part of the Vulkan loader etc. - and so tying this information directly to layers would’ve been awkward at best.

**Version History**

- Revision 1, 2018-11-05 (Tobias Hector)
  - Initial draft

**VK_EXT_transform_feedback**

**Name String**

- *VK_EXT_transform_feedback*

**Extension Type**

- Device extension

**Registered Extension Number**

- 29
Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Special Uses

• OpenGL / ES support
• D3D support
• Developer tools

Contact

• Piers Daniell @pdaniell-nv

Other Extension Metadata

Last Modified Date

2018-10-09

Contributors

• Baldur Karlsson, Valve
• Boris Zanin, Mobica
• Daniel Rakos, AMD
• Donald Scorgie, Imagination
• Henri Verbeet, CodeWeavers
• Jan-Harald Fredriksen, Arm
• Jason Ekstrand, Intel
• Jeff Bolz, NVIDIA
• Jesse Barker, Unity
• Jesse Hall, Google
• Pierre-Loup Griffais, Valve
• Philip Rebohle, DXVK
• Ruihao Zhang, Qualcomm
• Samuel Pitoiset, Valve
• Slawomir Grajewski, Intel
• Stu Smith, Imagination Technologies

Description

This extension adds transform feedback to the Vulkan API by exposing the SPIR-V
TransformFeedback and GeometryStreams capabilities to capture vertex, tessellation or geometry shader outputs to one or more buffers. It adds API functionality to bind transform feedback buffers to capture the primitives emitted by the graphics pipeline from SPIR-V outputs decorated for transform feedback. The transform feedback capture can be paused and resumed by way of storing and retrieving a byte counter. The captured data can be drawn again where the vertex count is derived from the byte counter without CPU intervention. If the implementation is capable, a vertex stream other than zero can be rasterized.

All these features are designed to match the full capabilities of OpenGL core transform feedback functionality and beyond. Many of the features are optional to allow base OpenGL ES GPUs to also implement this extension.

The primary purpose of the functionality exposed by this extension is to support translation layers from other 3D APIs. This functionality is not considered forward looking, and is not expected to be promoted to a KHR extension or to core Vulkan. Unless this is needed for translation, it is recommended that developers use alternative techniques of using the GPU to process and capture vertex data.

New Commands

- `vkCmdBeginQueryIndexedEXT`
- `vkCmdBeginTransformFeedbackEXT`
- `vkCmdBindTransformFeedbackBuffersEXT`
- `vkCmdDrawIndirectByteCountEXT`
- `vkCmdEndQueryIndexedEXT`
- `vkCmdEndTransformFeedbackEXT`

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceTransformFeedbackFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceTransformFeedbackPropertiesEXT`
- Extending `VkPipelineRasterizationStateCreateInfo`:
  - `VkPipelineRasterizationStateStreamCreateInfoEXT`

New Bitmasks

- `VkPipelineRasterizationStateStreamCreateFlagsEXT`

New Enum Constants

- `VK_EXT_TRANSFORM_FEEDBACK_EXTENSION_NAME`
- `VK_EXT_TRANSFORM_FEEDBACK_SPEC_VERSION`
• Extending VkAccessFlagBits:
  ◦ VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT
  ◦ VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT
  ◦ VK_ACCESS_TRANSFORM_FEEDBACK_WRITE_BIT_EXT

• Extending VkBufferUsageFlagBits:
  ◦ VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_BUFFER_BIT_EXT
  ◦ VK_BUFFER_USAGE_TRANSFORM_FEEDBACK_COUNTER_BUFFER_BIT_EXT

• Extending VkPipelineStageFlagBits:
  ◦ VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT

• Extending VkQueryType:
  ◦ VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_FEATURES_EXT
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_TRANSFORM_FEEDBACK_PROPERTIES_EXT
  ◦ VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_STREAM_CREATE_INFO_EXT

Issues

1) Should we include pause/resume functionality?

**RESOLVED**: Yes, this is needed to ease layering other APIs which have this functionality. To pause use `vkCmdEndTransformFeedbackEXT` and provide valid buffer handles in the `pCounterBuffers` array and offsets in the `pCounterBufferOffsets` array for the implementation to save the resume points. Then to resume use `vkCmdBeginTransformFeedbackEXT` with the previous `pCounterBuffers` and `pCounterBufferOffsets` values. Between the pause and resume there needs to be a memory barrier for the counter buffers with a source access of `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_WRITE_BIT_EXT` at pipeline stage `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT` to a destination access of `VK_ACCESS_TRANSFORM_FEEDBACK_COUNTER_READ_BIT_EXT` at pipeline stage `VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT`.

2) How does this interact with multiview?

**RESOLVED**: Transform feedback cannot be made active in a render pass with multiview enabled.

3) How should queries be done?

**RESOLVED**: There is a new query type `VK_QUERY_TYPE_TRANSFORM_FEEDBACK_STREAM_EXT`. A query pool created with this type will capture 2 integers - numPrimitivesWritten and numPrimitivesNeeded - for the specified vertex stream output from the last vertex processing stage. The vertex stream output queried is zero by default, but can be specified with the new `vkCmdBeginQueryIndexedEXT` and `vkCmdEndQueryIndexedEXT` commands.
Version History

- Revision 1, 2018-10-09 (Piers Daniell)
  - Internal revisions

VK_EXT_validation_cache

Name String

VK_EXT_validation_cache

Extension Type

Device extension

Registered Extension Number

161

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Cort Stratton cdwfs

Other Extension Metadata

Last Modified Date

2017-08-29

IP Status

No known IP claims.

Contributors

- Cort Stratton, Google
- Chris Forbes, Google

Description

This extension provides a mechanism for caching the results of potentially expensive internal validation operations across multiple runs of a Vulkan application. At the core is the VkValidationCacheEXT object type, which is managed similarly to the existing VkPipelineCache.

The new struct VkShaderModuleValidationCacheCreateInfoEXT can be included in the pNext chain at vkCreateShaderModule time. It contains a VkValidationCacheEXT to use when validating the VkShaderModule.
New Object Types

- VkValidationCacheEXT

New Commands

- vkCreateValidationCacheEXT
- vkDestroyValidationCacheEXT
- vkGetValidationCacheDataEXT
- vkMergeValidationCachesEXT

New Structures

- VkValidationCacheCreateInfoEXT
- Extending VkShaderModuleCreateInfo:
  - VkShaderModuleValidationCacheCreateInfoEXT

New Enums

- VkValidationCacheHeaderVersionEXT

New Bitmasks

- VkValidationCacheCreateFlagsEXT

New Enum Constants

- VK_EXT_VALIDATION_CACHE_EXTENSION_NAME
- VK_EXT_VALIDATION_CACHE_SPEC_VERSION
- Extending VkObjectType:
  - VK_OBJECT_TYPE_VALIDATION_CACHE_EXT
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_SHADER_MODULE_VALIDATION_CACHE_CREATE_INFO_EXT
  - VK_STRUCTURE_TYPE_VALIDATION_CACHE_CREATE_INFO_EXT

Version History

- Revision 1, 2017-08-29 (Cort Stratton)
  - Initial draft

VK_EXT_validation_features

Name String

VK_EXT_validation_features
Description

This extension provides the VkValidationFeaturesEXT struct that can be included in the pNext chain of the VkInstanceCreateInfo structure passed as the pCreateInfo parameter of vkCreateInstance. The structure contains an array of VkValidationFeatureEnableEXT enum values that enable specific validation features that are disabled by default. The structure also contains an array of VkValidationFeatureDisableEXT enum values that disable specific validation layer features that are enabled by default.
Note
The `VK_EXT_validation_features` extension subsumes all the functionality provided in the `VK_EXT_validation_flags` extension.

New Structures

- Extending `VkInstanceCreateInfo`:
  - `VkValidationFeaturesEXT`

New Enums

- `VkValidationFeatureDisableEXT`
- `VkValidationFeatureEnableEXT`

New Enum Constants

- `VK_EXT_VALIDATION_FEATURES_EXTENSION_NAME`
- `VK_EXT_VALIDATION_FEATURES_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_VALIDATION_FEATURES_EXT`

Version History

- Revision 1, 2018-11-14 (Karl Schultz)
  - Initial revision
- Revision 2, 2019-08-06 (Mark Lobodzinski)
  - Add Best Practices enable
- Revision 3, 2020-03-04 (Tony Barbour)
  - Add Debug Printf enable
- Revision 4, 2020-07-29 (John Zulauf)
  - Add Synchronization Validation enable

`VK_EXT_vertex_attribute_divisor`

Name String

- `VK_EXT_vertex_attribute_divisor`

Extension Type

- Device extension

Registered Extension Number

- 191
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Vikram Kushwaha @vkushwaha

Other Extension Metadata

Last Modified Date

2018-08-03

IP Status

No known IP claims.

Contributors

- Vikram Kushwaha, NVIDIA
- Jason Ekstrand, Intel

Description

This extension allows instance-rate vertex attributes to be repeated for certain number of instances instead of advancing for every instance when instanced rendering is enabled.

New Structures

- `VkVertexInputBindingDivisorDescriptionEXT`
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceVertexAttributeDivisorFeaturesEXT`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceVertexAttributeDivisorPropertiesEXT`
- Extending `VkPipelineVertexInputStateCreateInfo`:
  - `VkPipelineVertexInputDivisorStateCreateInfoEXT`

New Enum Constants

- `VK_EXT_VERTEX_ATTRIBUTE_DIVISOR_EXTENSION_NAME`
- `VK_EXT_VERTEX_ATTRIBUTE_DIVISOR_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_ATTRIBUTE_DIVISOR_FEATURES_EXT`
Issues

1) What is the effect of a non-zero value for `firstInstance`?

**RESOLVED:** The Vulkan API should follow the OpenGL convention and offset attribute fetching by `firstInstance` while computing vertex attribute offsets.

2) Should zero be an allowed divisor?

**RESOLVED:** Yes. A zero divisor means the vertex attribute is repeated for all instances.

Examples

To create a vertex binding such that the first binding uses instanced rendering and the same attribute is used for every 4 draw instances, an application could use the following set of structures:
const VkVertexInputBindingDivisorDescriptionEXT divisorDesc =
{
    0,
    4
};

const VkPipelineVertexInputDivisorStateCreateInfoEXT divisorInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_DIVISOR_STATE_CREATE_INFO_EXT, // sType
    NULL, // pNext
    1, // vertexBindingDivisorCount
    &divisorDesc // pVertexBindingDivisors
};

const VkVertexInputBindingDescription binding =
{
    0, // binding
    sizeof(Vertex), // stride
    VK_VERTEX_INPUT_RATE_INSTANCE // inputRate
};

const VkPipelineVertexInputStateCreateInfo viInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_CREATE_INFO, // sType
    &divisorInfo, // pNext
    ... //...
};

Version History

- Revision 1, 2017-12-04 (Vikram Kushwaha)
  ◦ First Version

- Revision 2, 2018-07-16 (Jason Ekstrand)
  ◦ Adjust the interaction between divisor and firstInstance to match the OpenGL convention.
  ◦ Disallow divisors of zero.

- Revision 3, 2018-08-03 (Vikram Kushwaha)
  ◦ Allow a zero divisor.
  ◦ Add a physical device features structure to query/enable this feature.

VK_EXT_vertex_input_dynamic_state
Name String

VK_EXT_vertex_input_dynamic_state

Extension Type

Device extension

Registered Extension Number

353

Revision

2

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Piers Daniell @pdaniell-nv

Other Extension Metadata

Last Modified Date

2020-08-21

IP Status

No known IP claims.

Contributors

• Jeff Bolz, NVIDIA
• Spencer Fricke, Samsung
• Stu Smith, AMD

Description

One of the states that contributes to the combinatorial explosion of pipeline state objects that need to be created, is the vertex input binding and attribute descriptions. By allowing them to be dynamic applications may reduce the number of pipeline objects they need to create.

This extension adds dynamic state support for what is normally static state in VkPipelineVertexInputStateCreateInfo.

New Commands

• vkCmdSetVertexInputEXT
New Structures

- VkVertexInputAttributeDescription2EXT
- VkVertexInputBindingDescription2EXT
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceVertexInputDynamicStateFeaturesEXT

New Enum Constants

- VK_EXT_VERTEX_INPUT_DYNAMIC_STATE_EXTENSION_NAME
- VK_EXT_VERTEX_INPUT_DYNAMIC_STATE_SPEC_VERSION
- Extending VkDynamicState:
  - VK_DYNAMIC_STATE_VERTEX_INPUT_EXT
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VERTEX_INPUT_DYNAMIC_STATE_FEATURES_EXT
  - VK_STRUCTURE_TYPE_VERTEX_INPUT_ATTRIBUTE_DESCRIPTION_2_EXT
  - VK_STRUCTURE_TYPE_VERTEX_INPUT_BINDING_DESCRIPTION_2_EXT

Version History

- Revision 2, 2020-11-05 (Piers Daniell)
  - Make VkVertexInputBindingDescription2EXT extensible
  - Add new VkVertexInputAttributeDescription2EXT struct for the pVertexAttributeDescriptions parameter to vkCmdSetVertexInputEXT so it is also extensible
- Revision 1, 2020-08-21 (Piers Daniell)
  - Internal revisions

VK_EXT_ycbcr_2plane_444_formats

Name String

VK_EXT_ycbcr_2plane_444_formats

Extension Type

Device extension

Registered Extension Number

331

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
• Requires VK_KHR_sampler_ycbcr_conversion

Contact
• Tony Zlatinski

Other Extension Metadata

Last Modified Date
2020-07-28

IP Status
No known IP claims.

Contributors
• Piers Daniell, NVIDIA
• Ping Liu, Intel

Description
This extension adds some Y′CbCr formats that are in common use for video encode and decode, but were not part of the VK_KHR_sampler_ycbcr_conversion extension.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceYcbcr2Plane444FormatsFeaturesEXT

New Enum Constants
• VK_EXT_YCBCR_2PLANE_444_FORMATS_EXTENSION_NAME
• VK_EXT_YCBCR_2PLANE_444_FORMATS_SPEC_VERSION
• Extending VkFormat:
  ◦ VK_FORMAT_G10X6_B10X6R10X6_2PLANE_444_UNORM_3PACK16_EXT
  ◦ VK_FORMAT_G12X4_B12X4R12X4_2PLANE_444_UNORM_3PACK16_EXT
  ◦ VK_FORMAT_G16_B16R16_2PLANE_444_UNORM_EXT
  ◦ VK_FORMAT_G8_B8R8_2PLANE_444_UNORM_EXT
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_YCBCR_2_PLANE_444_FORMATS_FEATURES_EXT

Version History
• Revision 1, 2020-03-08 (Piers Daniell)
  ◦ Initial draft
VK_EXT_ycbcr_image_arrays

Name String
  VK_EXT_ycbcr_image_arrays

Extension Type
  Device extension

Registered Extension Number
  253

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_sampler_ycbcr_conversion

Contact
  • Piers Daniell pdaniell-nv

Other Extension Metadata

Last Modified Date
  2019-01-15

Contributors
  • Piers Daniell, NVIDIA

Description
This extension allows images of a format that requires $Y'\text{C}_b\text{C}_r$ conversion to be created with multiple array layers, which is otherwise restricted.

New Structures
  • Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
    ◦ VkPhysicalDeviceYcbcrImageArraysFeaturesEXT

New Enum Constants
  • VK_EXT_YCBCR_IMAGE_ARRAYS_EXTENSION_NAME
  • VK_EXT_YCBCR_IMAGE_ARRAYS_SPEC_VERSION
  • Extending VkStructureType:
    ◦ VK_STRUCTURE>Type_PHYSICAL_DEVICE_YCBCR_IMAGE_ARRAYS_FEATURES_EXT
Version History

• Revision 1, 2019-01-15 (Piers Daniell)
  ◦ Initial revision

VK_AMD_buffer_marker

Name String
  VK_AMD_buffer_marker

Extension Type
  Device extension

Registered Extension Number
  180

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0

Special Use
  • Developer tools

Contact
  • Daniel Rakos @drakos-amd

Other Extension Metadata

Last Modified Date
  2018-01-26

IP Status
  No known IP claims.

Contributors
  • Matthaeus G. Chajdas, AMD
  • Jaakko Konttinen, AMD
  • Daniel Rakos, AMD

Description

This extension adds a new operation to execute pipelined writes of small marker values into a VkBuffer object.

The primary purpose of these markers is to facilitate the development of debugging tools for
tracking which pipelined command contributed to device loss.

**New Commands**

- `vkCmdWriteBufferMarkerAMD`

**New Enum Constants**

- `VK_AMD_BUFFER_MARKER_EXTENSION_NAME`
- `VK_AMD_BUFFER_MARKER_SPEC_VERSION`

**Examples**

None.

**Version History**

- Revision 1, 2018-01-26 (Jaakko Konttinen)
  - Initial revision

**VK_AMD_device_coherent_memory**

**Name String**

`VK_AMD_device_coherent_memory`

**Extension Type**

Device extension

**Registered Extension Number**

230

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Tobias Hector @tobski

**Other Extension Metadata**

**Last Modified Date**

2019-02-04

**Contributors**

- Ping Fu, AMD
**Description**

This extension adds the device coherent and device uncached memory types. Any device accesses to device coherent memory are automatically made visible to any other device access. Device uncached memory indicates to applications that caches are disabled for a particular memory type, which guarantees device coherence.

Device coherent and uncached memory are expected to have lower performance for general access than non-device coherent memory, but can be useful in certain scenarios; particularly so for debugging.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceCoherentMemoryFeaturesAMD`

**New Enum Constants**

- `VK_AMD_DEVICE_COHERENT_MEMORY_EXTENSION_NAME`
- `VK_AMD_DEVICE_COHERENT_MEMORY_SPEC_VERSION`

Extending `VkMemoryPropertyFlagBits`:

- `VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD`
- `VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD`

Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COHERENT_MEMORY_FEATURES_AMD`

**Version History**

- Revision 1, 2019-02-04 (Tobias Hector)
  - Initial revision

**VK_AMD_display_native_hdr**

**Name String**

`VK_AMD_display_native_hdr`

**Extension Type**

Device extension

**Registered Extension Number**

214
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2
- Requires VK_KHR_get_surface_capabilities2
- Requires VK_KHR_swapchain

Contact

- Matthaeus G. Chajdas

Other Extension Metadata

Last Modified Date
2018-12-18

IP Status
No known IP claims.

Contributors

- Matthaeus G. Chajdas, AMD
- Aaron Hagan, AMD
- Aric Cyr, AMD
- Timothy Lottes, AMD
- Derrick Owens, AMD
- Daniel Rakos, AMD

Description

This extension introduces the following display native HDR features to Vulkan:

- A new VkColorSpaceKHR enum for setting the native display colorspace. For example, this color space would be set by the swapchain to use the native color space in Freesync2 displays.
- Local dimming control

New Commands

- vkSetLocalDimmingAMD

New Structures

- Extending VkSurfaceCapabilities2KHR:
  - VkDisplayNativeHdrSurfaceCapabilitiesAMD
• Extending VkSwapchainCreateInfoKHR:
  ◦ VkSwapchainDisplayNativeHdrCreateInfoAMD

New Enum Constants

• VK_AMD_DISPLAY_NATIVE_HDR_EXTENSION_NAME
• VK_AMD_DISPLAY_NATIVE_HDR_SPEC_VERSION

• Extending VkColorSpaceKHR:
  ◦ VK_COLOR_SPACE_DISPLAY_NATIVE_AMD

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_DISPLAY_NATIVE_HDR_SURFACE_CAPABILITIES_AMD
  ◦ VK_STRUCTURE_TYPE_SWAPCHAIN_DISPLAY_NATIVE_HDR_CREATE_INFO_AMD

Issues

None.

Examples

None.

Version History

• Revision 1, 2018-12-18 (Daniel Rakos)
  ◦ Initial revision

VK_AMD_gcn_shader

Name String

VK_AMD_gcn_shader

Extension Type

Device extension

Registered Extension Number

26

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Dominik Witczak dominiwitzackam
Other Extension Metadata

Last Modified Date
2016-05-30

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_AMD_gcn_shader

Contributors
• Dominik Witczak, AMD
• Daniel Rakos, AMD
• Rex Xu, AMD
• Graham Sellers, AMD

Description
This extension adds support for the following SPIR-V extension in Vulkan:

• SPV_AMD_gcn_shader

New Enum Constants
• VK_AMD_GCN_SHADER_EXTENSION_NAME
• VK_AMD_GCN_SHADER_SPEC_VERSION

Version History
• Revision 1, 2016-05-30 (Dominik Witczak)
  ◦ Initial draft

VK_AMD_memory_overallocation_behavior

Name String
VK_AMD_memory_overallocation_behavior

Extension Type
Device extension

Registered Extension Number
190

Revision
1
Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Martin Dinkov mdinkov

Other Extension Metadata

Last Modified Date

2018-09-19

IP Status

No known IP claims.

Contributors

- Martin Dinkov, AMD
- Matthaeus Chajdas, AMD
- Daniel Rakos, AMD
- Jon Campbell, AMD

Description

This extension allows controlling whether explicit overallocation beyond the device memory heap sizes (reported by VkPhysicalDeviceMemoryProperties) is allowed or not. Overallocation may lead to performance loss and is not supported for all platforms.

New Structures

- Extending VkDeviceCreateInfo:
  - VkDeviceMemoryOverallocationCreateInfoAMD

New Enums

- VkMemoryOverallocationBehaviorAMD

New Enum Constants

- VK_AMD_MEMORY_OVERALLOCATION_BEHAVIOR_EXTENSION_NAME
- VK_AMD_MEMORY_OVERALLOCATION_BEHAVIOR_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DEVICE_MEMORY_OVERALLOCATION_CREATE_INFO_AMD

Version History

- Revision 1, 2018-09-19 (Martin Dinkov)
VK_AMD_mixed_attachment_samples

Name String
VK_AMD_mixed_attachment_samples

Extension Type
Device extension

Registered Extension Number
137

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Matthaeus G. Chajdas 📡anteru

Other Extension Metadata

Last Modified Date
2017-07-24

Contributors
• Mais Alnasser, AMD
• Matthaeus G. Chajdas, AMD
• Maciej Jesionowski, AMD
• Daniel Rakos, AMD

Description
This extension enables applications to use multisampled rendering with a depth/stencil sample count that is larger than the color sample count. Having a depth/stencil sample count larger than the color sample count allows maintaining geometry and coverage information at a higher sample rate than color information. All samples are depth/stencil tested, but only the first color sample count number of samples get a corresponding color output.

New Enum Constants
• VK_AMD_MIXED_ATTACHMENT_SAMPLES_EXTENSION_NAME
• VK_AMD_MIXED_ATTACHMENT_SAMPLES_SPEC_VERSION
**Issues**

None.

**Version History**

- Revision 1, 2017-07-24 (Daniel Rakos)
  - Internal revisions

**VK_AMD_pipeline_compiler_control**

**Name String**

VK_AMD_pipeline_compiler_control

**Extension Type**

Device extension

**Registered Extension Number**

184

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Matthaeus G. Chajdas

**Other Extension Metadata**

**Last Modified Date**

2019-07-26

**IP Status**

No known IP claims.

**Contributors**

- Matthaeus G. Chajdas, AMD
- Daniel Rakos, AMD
- Maciej Jesionowski, AMD
- Tobias Hector, AMD

**Description**

This extension introduces VkPipelineCompilerControlCreateInfoAMD structure that can be chained to a pipeline’s create info to specify additional flags that affect pipeline compilation.
New Structures

- Extending VkGraphicsPipelineCreateInfo, VkComputePipelineCreateInfo:
  - VkPipelineCompilerControlCreateInfoAMD

New Enums

- VkPipelineCompilerControlFlagBitsAMD

New Bitmasks

- VkPipelineCompilerControlFlagsAMD

New Enum Constants

- VK_AMD_PIPELINE_COMPILER_CONTROL_EXTENSION_NAME
- VK_AMD_PIPELINE_COMPILER_CONTROL_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PIPELINE_COMPILER_CONTROL_CREATE_INFO_AMD

Issues

None.

Examples

None.

Version History

- Revision 1, 2019-07-26 (Tobias Hector)
  - Initial revision.

VK_AMD_rasterization_order

Name String

VK_AMD_rasterization_order

Extension Type

Device extension

Registered Extension Number

19

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Daniel Rakos @drakos-amd

Other Extension Metadata

Last Modified Date

2016-04-25

IP Status

No known IP claims.

Contributors

- Matthaeus G. Chajdas, AMD
- Jaakko Konttinen, AMD
- Daniel Rakos, AMD
- Graham Sellers, AMD
- Dominik Witczak, AMD

Description

This extension introduces the possibility for the application to control the order of primitive rasterization. In unextended Vulkan, the following stages are guaranteed to execute in API order:

- depth bounds test
- stencil test, stencil op, and stencil write
- depth test and depth write
- occlusion queries
- blending, logic op, and color write

This extension enables applications to opt into a relaxed, implementation defined primitive rasterization order that may allow better parallel processing of primitives and thus enabling higher primitive throughput. It is applicable in cases where the primitive rasterization order is known to not affect the output of the rendering or any differences caused by a different rasterization order are not a concern from the point of view of the application's purpose.

A few examples of cases when using the relaxed primitive rasterization order would not have an effect on the final rendering:

- If the primitives rendered are known to not overlap in framebuffer space.
- If depth testing is used with a comparison operator of `VK_COMPARE_OP_LESS, VK_COMPARE_OP_LESS_OR_EQUAL, VK_COMPARE_OP_GREATER, or VK_COMPARE_OP_GREATER_OR_EQUAL`, and the primitives rendered are known to not overlap in clip space.
• If depth testing is not used and blending is enabled for all attachments with a commutative blend operator.

New Structures

• Extending VkPipelineRasterizationStateCreateInfo:
  ◦ VkPipelineRasterizationStateRasterizationOrderAMD

New Enums

• VkRasterizationOrderAMD

New Enum Constants

• VK_AMD_RASTERIZATION_ORDER_EXTENSION_NAME
• VK_AMD_RASTERIZATION_ORDER_SPEC_VERSION

Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_RASTERIZATION_ORDER_AMD

Issues

1) How is this extension useful to application developers?

**RESOLVED:** Allows them to increase primitive throughput for cases when strict API order rasterization is not important due to the nature of the content, the configuration used, or the requirements towards the output of the rendering.

2) How does this extension interact with content optimizations aiming to reduce overdraw by appropriately ordering the input primitives?

**RESOLVED:** While the relaxed rasterization order might somewhat limit the effectiveness of such content optimizations, most of the benefits of it are expected to be retained even when the relaxed rasterization order is used, so applications **should** still apply these optimizations even if they intend to use the extension.

3) Are there any guarantees about the primitive rasterization order when using the new relaxed mode?

**RESOLVED:** No. In this case the rasterization order is completely implementation dependent, but in practice it is expected to partially still follow the order of incoming primitives.

4) Does the new relaxed rasterization order have any adverse effect on repeatability and other invariance rules of the API?

**RESOLVED:** Yes, in the sense that it extends the list of exceptions when the repeatability requirement does not apply.
Examples
None

Issues
None

Version History

• Revision 1, 2016-04-25 (Daniel Rakos)
  ◦ Initial draft.

VK_AMD_shader_ballot

Name String
  VK_AMD_shader_ballot

Extension Type
  Device extension

Registered Extension Number
  38

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0

Contact
  • Dominik Witczak @dominikwitczakamd

Other Extension Metadata

Last Modified Date
  2016-09-19

IP Status
  No known IP claims.

Interactions and External Dependencies
  • This extension requires SPV_AMD_shader_ballot

Contributors
  • Qun Lin, AMD
  • Graham Sellers, AMD
Description

This extension adds support for the following SPIR-V extension in Vulkan:

- SPV_AMD_shader_ballot

New Enum Constants

- VK_AMD_SHADER_BALLOT_EXTENSION_NAME
- VK_AMD_SHADER_BALLOT_SPEC_VERSION

Version History

- Revision 1, 2016-09-19 (Dominik Witczak)
  - Initial draft

VK_AMD_shader_core_properties

Name String

- VK_AMD_shader_core_properties

Extension Type

- Device extension

Registered Extension Number

- 186

Revision

- 2

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Contact

- Martin Dinkov mdinkov

Other Extension Metadata

Last Modified Date

- 2019-06-25
Description

This extension exposes shader core properties for a target physical device through the `VK_KHR_get_physical_device_properties2` extension. Please refer to the example below for proper usage.

New Structures

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceShaderCorePropertiesAMD`

New Enum Constants

- `VK_AMD_SHADER_CORE_PROPERTIES_EXTENSION_NAME`
- `VK_AMD_SHADER_CORE_PROPERTIES_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_AMD`

Examples

This example retrieves the shader core properties for a physical device.

```c
extern VkInstance instance;

PFN_vkGetPhysicalDeviceProperties2 pfnVkGetPhysicalDeviceProperties2 =
  reinterpret_cast<PFN_vkGetPhysicalDeviceProperties2>(
vkGetInstanceProcAddr(instance, "vkGetPhysicalDeviceProperties2"));

VkPhysicalDeviceProperties2             general_props;
VkPhysicalDeviceShaderCorePropertiesAMD shader_core_properties;

shader_core_properties.pNext = nullptr;
shader_core_properties.sType =
  VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_AMD;

general_props.pNext = &shader_core_properties;
general_props.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;

// After this call, shader_core_properties has been populated
pfnVkGetPhysicalDeviceProperties2(device, &general_props);
```
printf("Number of shader engines: %d\n",
    m_shader_core_properties.shader_engine_count = 
    shader_core_properties.shaderEngineCount;
printf("Number of shader arrays: %d\n",
    m_shader_core_properties.shader_arrays_per_engine_count = 
    shader_core_properties.shaderArraysPerEngineCount;
printf("Number of CUs per shader array: %d\n",
    m_shader_core_properties.compute_units_per_shader_array = 
    shader_core_properties.computeUnitsPerShaderArray;
printf("Number of SIMDs per compute unit: %d\n",
    m_shader_core_properties.simd_per_compute_unit = 
    shader_core_properties.simdPerComputeUnit;
printf("Number of wavefront slots in each SIMD: %d\n",
    m_shader_core_properties.wavefronts_per_simd = 
    shader_core_properties.wavefrontsPerSimd;
printf("Number of threads per wavefront: %d\n",
    m_shader_core_properties.wavefront_size = 
    shader_core_properties.wavefrontSize;
printf("Number of physical SGPRs per SIMD: %d\n",
    m_shader_core_properties.sgprs_per_simd = 
    shader_core_properties.sgprsPerSimd;
printf("Minimum number of SGPRs that can be allocated by a wave: %d\n",
    m_shader_core_properties.min_sgpr_allocation = 
    shader_core_properties.minSgprAllocation;
printf("Number of available SGPRs: %d\n",
    m_shader_core_properties.max_sgpr_allocation = 
    shader_core_properties.maxSgprAllocation;
printf("SGPRs are allocated in groups of this size: %d\n",
    m_shader_core_properties.sgpr_allocation_granularity = 
    shader_core_properties.sgprAllocationGranularity;
printf("Number of physical VGPRs per SIMD: %d\n",
    m_shader_core_properties.vgprs_per_simd = 
    shader_core_properties.vgprsPerSimd;
printf("Minimum number of VGPRs that can be allocated by a wave: %d\n",
    m_shader_core_properties.min_vgpr_allocation = 
    shader_core_properties.minVgprAllocation;
printf("Number of available VGPRs: %d\n",
    m_shader_core_properties.max_vgpr_allocation = 
    shader_core_properties.maxVgprAllocation;
printf("VGPRs are allocated in groups of this size: %d\n",
    m_shader_core_properties.vgpr_allocation_granularity = 
    shader_core_properties.vgprAllocationGranularity;

Version History

• Revision 2, 2019-06-25 (Matthaeus G. Chajdas)
  ◦ Clarified the meaning of a few fields.
• Revision 1, 2018-02-15 (Martin Dinkov)
VK_AMD_shader_core_properties2

Name String
VK_AMD_shader_core_properties2

Extension Type
Device extension

Registered Extension Number
228

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_AMD_shader_core_properties

Contact
• Matthaeus G. Chajdas anteru

Other Extension Metadata

Last Modified Date
2019-07-26

IP Status
No known IP claims.

Contributors
• Matthaeus G. Chajdas, AMD
• Tobias Hector, AMD

Description
This extension exposes additional shader core properties for a target physical device through the VK_KHR_get_physical_device_properties2 extension.

New Structures
• Extending VkPhysicalDeviceProperties2:
  • VkPhysicalDeviceShaderCoreProperties2AMD
New Enums

- VkShaderCorePropertiesFlagBitsAMD

New Bitmasks

- VkShaderCorePropertiesFlagsAMD

New Enum Constants

- VK_AMD_SHADER_CORE_PROPERTIES_2_EXTENSION_NAME
- VK_AMD_SHADER_CORE_PROPERTIES_2_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_CORE_PROPERTIES_2_AMD

Examples

None.

Version History

- Revision 1, 2019-07-26 (Matthaeus G. Chajdas)
  - Initial draft.

VK_AMD_shader_explicit_vertex_parameter

Name String

VK_AMD_shader_explicit_vertex_parameter

Extension Type

Device extension

Registered Extension Number

22

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Qun Lin (linqun)

Other Extension Metadata
Last Modified Date
2016-05-10

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_AMD_shader_explicit_vertex_parameter

Contributors
• Matthaeus G. Chajdas, AMD
• Qun Lin, AMD
• Daniel Rakos, AMD
• Graham Sellers, AMD
• Rex Xu, AMD

Description
This extension adds support for the following SPIR-V extension in Vulkan:

• SPV_AMD_shader_explicit_vertex_parameter

New Enum Constants
• VK_AMD_SHADER_EXPLICIT_VERTEX_PARAMETER_EXTENSION_NAME
• VK_AMD_SHADER_EXPLICIT_VERTEX_PARAMETER_SPEC_VERSION

Version History
• Revision 1, 2016-05-10 (Daniel Rakos)
  ◦ Initial draft

VK_AMD_shader_fragment_mask

Name String
VK_AMD_shader_fragment_mask

Extension Type
Device extension

Registered Extension Number
138

Revision
1
Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Aaron Hagan @AaronHaganAMD

Other Extension Metadata

Last Modified Date

2017-08-16

IP Status

No known IP claims.

Interactions and External Dependencies

- This extension requires SPV_AMD_shader_fragment_mask

Contributors

- Aaron Hagan, AMD
- Daniel Rakos, AMD
- Timothy Lottes, AMD

Description

This extension provides efficient read access to the fragment mask in compressed multisampled color surfaces. The fragment mask is a lookup table that associates color samples with color fragment values.

From a shader, the fragment mask can be fetched with a call to fragmentMaskFetchAMD, which returns a single uint where each subsequent four bits specify the color fragment index corresponding to the color sample, starting from the least significant bit. For example, when eight color samples are used, the color fragment index for color sample 0 will be in bits 0-3 of the fragment mask, for color sample 7 the index will be in bits 28-31.

The color fragment for a particular color sample may then be fetched with the corresponding fragment mask value using the fragmentFetchAMD shader function.

New Enum Constants

- VK_AMD_SHADER_FRAGMENT_MASK_EXTENSION_NAME
- VK_AMD_SHADER_FRAGMENT_MASK_SPEC_VERSION

New SPIR-V Capabilities

- FragmentMaskAMD
Examples
This example shows a shader that queries the fragment mask from a multisampled compressed surface and uses it to query fragment values.

```glsl
#version 450 core

#extension GL_AMD_shader_fragment_mask: enable

layout(binding = 0) uniform sampler2DMS s2DMS;
lazy(binding = 1) uniform isampler2DMSArray is2DMSArray;
layout(binding = 2, input_attachment_index = 0) uniform usubpassInputMS usubpassMS;
layout(location = 0) out vec4 fragColor;

void main()
{
    vec4 fragOne = vec4(0.0);

    uint fragMask = fragmentMaskFetchAMD(s2DMS, ivec2(2, 3));
    uint fragIndex = (fragMask & 0xF0) >> 4;
    fragOne += fragmentFetchAMD(s2DMS, ivec2(2, 3), 1);

    fragMask = fragmentMaskFetchAMD(is2DMSArray, ivec3(2, 3, 1));
    fragIndex = (fragMask & 0xF0) >> 4;
    fragOne += fragmentFetchAMD(is2DMSArray, ivec3(2, 3, 1), fragIndex);

    fragMask = fragmentMaskFetchAMD(usubpassMS);
    fragIndex = (fragMask & 0xF0) >> 4;
    fragOne += fragmentFetchAMD(usubpassMS, fragIndex);

    fragColor = fragOne;
}
```

Version History

- Revision 1, 2017-08-16 (Aaron Hagan)
  - Initial draft

VK_AMD_shader_image_load_store_lod

Name String

`VK_AMD_shader_image_load_store_lod`

Extension Type

Device extension
Registered Extension Number
47

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Dominik Witczak dominikwitczakamd

Other Extension Metadata

Last Modified Date
2017-08-21

Interactions and External Dependencies
• This extension requires SPV_AMD_shader_image_load_store_lod
• This extension provides API support for GL_AMD_shader_image_load_store_lod

IP Status
No known IP claims.

Contributors
• Dominik Witczak, AMD
• Qun Lin, AMD
• Rex Xu, AMD

Description
This extension adds support for the following SPIR-V extension in Vulkan:

• SPV_AMD_shader_image_load_store_lod

New Enum Constants

• VK_AMD_SHADER_IMAGE_LOAD_STORE_LOD_EXTENSION_NAME
• VK_AMD_SHADER_IMAGE_LOAD_STORE_LOD_SPEC_VERSION

Version History
• Revision 1, 2017-08-21 (Dominik Witczak)
  ◦ Initial draft
VK_AMD_shader_info

Name String
VK_AMD_shader_info

Extension Type
Device extension

Registered Extension Number
43

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Special Use
• Developer tools

Contact
• Jaakko Konttinen @jaakkoamd

Other Extension Metadata

Last Modified Date
2017-10-09

IP Status
No known IP claims.

Contributors
• Jaakko Konttinen, AMD

Description
This extension adds a way to query certain information about a compiled shader which is part of a pipeline. This information may include shader disassembly, shader binary and various statistics about a shader’s resource usage.

While this extension provides a mechanism for extracting this information, the details regarding the contents or format of this information are not specified by this extension and may be provided by the vendor externally.

Furthermore, all information types are optionally supported, and users should not assume every implementation supports querying every type of information.
New Commands

• vkGetShaderInfoAMD

New Structures

• VkShaderResourceUsageAMD
• VkShaderStatisticsInfoAMD

New Enums

• VkShaderInfoTypeAMD

New Enum Constants

• VK_AMD_SHADER_INFO_EXTENSION_NAME
• VK_AMD_SHADER_INFO_SPEC_VERSION

Examples

This example extracts the register usage of a fragment shader within a particular graphics pipeline:

```c
extern VkDevice device;
extern VkPipeline gfxPipeline;

PFN_vkGetShaderInfoAMD pfnGetShaderInfoAMD = (PFN_vkGetShaderInfoAMD)
vkGetDeviceProcAddr(device, "vkGetShaderInfoAMD");

VkShaderStatisticsInfoAMD statistics = {};

size_t dataSize = sizeof(statistics);

if (pfnGetShaderInfoAMD(device, gfxPipeline,
VK_SHADER_STAGE_FRAGMENT_BIT,
VK_SHADER_INFO_TYPE_STATISTICS_AMD,
&dataSize,
&statistics) == VK_SUCCESS)
{
    printf("VGPR usage: %d\n", statistics.resourceUsage.numUsedVgprs);
    printf("SGPR usage: %d\n", statistics.resourceUsage.numUsedSgprs);
}
```

The following example continues the previous example by subsequently attempting to query and print shader disassembly about the fragment shader:
// Query disassembly size (if available)
if (pfnGetShaderInfoAMD(device,
    gfxPipeline,
    VK_SHADER_STAGE_FRAGMENT_BIT,
    VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD,
    &dataSize,
    nullptr) == VK_SUCCESS)
{
    printf("Fragment shader disassembly:\n");

    void* disassembly = malloc(dataSize);

    // Query disassembly and print
    if (pfnGetShaderInfoAMD(device,
            gfxPipeline,
            VK_SHADER_STAGE_FRAGMENT_BIT,
            VK_SHADER_INFO_TYPE_DISASSEMBLY_AMD,
            &dataSize,
            disassembly) == VK_SUCCESS)
    {
        printf((char*)disassembly);
    }

    free(disassembly);
}

Version History

- Revision 1, 2017-10-09 (Jaakko Konttinen)
  - Initial revision

VK_AMD_shader_trinary_minmax

Name String

VK_AMD_shader_trinary_minmax

Extension Type

Device extension

Registered Extension Number

21

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
Contact
• Qun Lin linqun

Other Extension Metadata

Last Modified Date
2016-05-10

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_AMD_shader_trinary_minmax

Contributors
• Matthaeus G. Chajdas, AMD
• Qun Lin, AMD
• Daniel Rakos, AMD
• Graham Sellers, AMD
• Rex Xu, AMD

Description
This extension adds support for the following SPIR-V extension in Vulkan:
• SPV_AMD_shader_trinary_minmax

New Enum Constants
• VK_AMD_SHADER_TRINARY_MINMAX_EXTENSION_NAME
• VK_AMD_SHADER_TRINARY_MINMAX_SPEC_VERSION

Version History
• Revision 1, 2016-05-10 (Daniel Rakos)
  ◦ Initial draft

VK_AMD_texture_gather_bias_lod

Name String
VK_AMD_texture_gather_bias_lod

Extension Type
Device extension
This extension adds two related features.

Firstly, support for the following SPIR-V extension in Vulkan is added:

- `SPV_AMD_texture_gather_bias_lod`

Secondly, the extension allows the application to query which formats can be used together with the new function prototypes introduced by the SPIR-V extension.
New Structures

• Extending VkImageFormatProperties2:
  ◦ VkTextureLODGatherFormatPropertiesAMD

New Enum Constants

• VK_AMD_TEXTURE_GATHER_BIAS_LOD_EXTENSION_NAME
• VK_AMD_TEXTURE_GATHER_BIAS_LOD_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD

New SPIR-V Capabilities

• ImageGatherBiasLodAMD

Examples
struct VkTextureLODGatherFormatPropertiesAMD
{
    VkStructureType sType;
    const void* pNext;
    VkBool32 supportsTextureGatherLODBiasAMD;
};

    // How to detect if an image format can be used with the new function prototypes.
VkPhysicalDeviceImageFormatInfo2 formatInfo;
VkImageFormatProperties2 formatProps;
VkTextureLODGatherFormatPropertiesAMD textureLODGatherSupport;

    textureLODGatherSupport.sType = VK_STRUCTURE_TYPE_TEXTURE_LOD_GATHER_FORMAT_PROPERTIES_AMD;
    textureLODGatherSupport.pNext = nullptr;

    formatInfo.sType  = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2;
    formatInfo.pNext  = nullptr;
    formatInfo.format = ...;
    formatInfo.type   = ...;
    formatInfo.tiling = ...;
    formatInfo.usage  = ...;
    formatInfo.flags  = ...;

    formatProps.sType = VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2;
    formatProps.pNext = &textureLODGatherSupport;

vkGetPhysicalDeviceImageFormatProperties2(physical_device, &formatInfo, &formatProps);

    if (textureLODGatherSupport.supportsTextureGatherLODBiasAMD == VK_TRUE)
    {
        // physical device supports SPV_AMD_texture_gather_bias_lod for the specified
        // format configuration.
    }
    else
    {
        // physical device does not support SPV_AMD_texture_gather_bias_lod for the
        // specified format configuration.
    }

Version History

• Revision 1, 2017-03-21 (Dominik Witczak)
  ◦ Initial draft
VK_ANDROID_external_memory_android_hardware_buffer

Name String
VK_ANDROID_external_memory_android_hardware_buffer

Extension Type
Device extension

Registered Extension Number
130

Revision
3

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_sampler_ycbcr_conversion
- Requires VK_KHR_external_memory
- Requires VK_EXT_queue_family_foreign
- Requires VK_KHR_dedicated_allocation

Contact
- Jesse Hall https://critsec

Other Extension Metadata

Last Modified Date
2019-08-27

IP Status
No known IP claims.

Contributors
- Ray Smith, ARM
- Chad Versace, Google
- Jesse Hall, Google
- Tobias Hector, Imagination
- James Jones, NVIDIA
- Tony Zlatinski, NVIDIA
- Matthew Netsch, Qualcomm
- Andrew Garrard, Samsung
This extension enables an application to import Android AHardwareBuffer objects created outside of the Vulkan device into Vulkan memory objects, where they can be bound to images and buffers. It also allows exporting an AHardwareBuffer from a Vulkan memory object for symmetry with other operating systems. But since not all AHardwareBuffer usages and formats have Vulkan equivalents, exporting from Vulkan provides strictly less functionality than creating the AHardwareBuffer externally and importing it.

Some AHardwareBuffer images have implementation-defined external formats that may not correspond to Vulkan formats. Sampler Y'CbCr conversion can be used to sample from these images and convert them to a known color space.

New Base Types

- AHardwareBuffer

New Commands

- vkGetAndroidHardwareBufferPropertiesANDROID
- vkGetMemoryAndroidHardwareBufferANDROID

New Structures

- VkAndroidHardwareBufferPropertiesANDROID
- VkMemoryGetAndroidHardwareBufferInfoANDROID
- Extending VkAndroidHardwareBufferPropertiesANDROID:
  - VkAndroidHardwareBufferFormatPropertiesANDROID
- Extending VkImageCreateInfo, VkSamplerYcbcrConversionCreateInfo:
  - VkExternalFormatANDROID
- Extending VkImageFormatProperties2:
  - VkAndroidHardwareBufferUsageANDROID
- Extending VkMemoryAllocateInfo:
  - VkImportAndroidHardwareBufferInfoANDROID

New Enum Constants

- VK_ANDROID_EXTERNAL_MEMORY_ANDROID_HARDWARE_BUFFER_EXTENSION_NAME
- VK_ANDROID_EXTERNAL_MEMORY_ANDROID_HARDWARE_BUFFER_SPEC_VERSION
- Extending VkExternalMemoryHandleTypeFlagBits:
  - VK_EXTERNAL_MEMORY_HANDLE_TYPE_ANDROID_HARDWARE_BUFFER_BIT_ANDROID
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_ANDROID_HARDWARE_BUFFER_FORMAT_PROPERTIES_ANDROID
Issues

1) Other external memory objects are represented as weakly-typed handles (e.g. Win32 \texttt{HANDLE} or POSIX file descriptor), and require a handle type parameter along with handles. \texttt{AHardwareBuffer} is strongly typed, so naming the handle type is redundant. Does symmetry justify adding handle type parameters/fields anyway?

**RESOLVED:** No. The handle type is already provided in places that treat external memory objects generically. In the places we would add it, the application code that would have to provide the handle type value is already dealing with \texttt{AHardwareBuffer}-specific commands/structures; the extra symmetry would not be enough to make that code generic.

2) The internal layout and therefore size of a \texttt{AHardwareBuffer} image may depend on native usage flags that do not have corresponding Vulkan counterparts. Do we provide this info to \texttt{vkCreateImage} somehow, or allow the allocation size reported by \texttt{vkGetImageMemoryRequirements} to be approximate?

**RESOLVED:** Allow the allocation size to be unspecified when allocating the memory. It has to work this way for exported image memory anyway, since \texttt{AHardwareBuffer} allocation happens in \texttt{vkAllocateMemory}, and internally is performed by a separate HAL, not the Vulkan implementation itself. There is a similar issue with \texttt{vkGetImageSubresourceLayout}: the layout is determined by the allocator HAL, so it is not known until the image is bound to memory.

3) Should the result of sampling an external-format image with the suggested Y'CbCr conversion parameters yield the same results as using a \texttt{samplerExternalOES} in OpenGL ES?

**RESOLVED:** This would be desirable, so that apps converting from OpenGL ES to Vulkan could get the same output given the same input. But since sampling and conversion from Y'CbCr images is so loosely defined in OpenGL ES, multiple implementations do it in a way that doesn’t conform to Vulkan’s requirements. Modifying the OpenGL ES implementation would be difficult, and would change the output of existing unmodified applications. Changing the output only for applications that are being modified gives developers the chance to notice and mitigate any problems. Implementations are encouraged to minimize differences as much as possible without causing compatibility problems for existing OpenGL ES applications or violating Vulkan requirements.

4) Should an \texttt{AHardwareBuffer} with \texttt{AHARDWAREBUFFER_USAGE_CPU_\*} usage be mappable in Vulkan? Should it be possible to export an \texttt{AHardwareBuffers} with such usage?

**RESOLVED:** Optional, and mapping in Vulkan is not the same as \texttt{AHardwareBuffer_lock}. The semantics of these are different: mapping in memory is persistent, just gives a raw view of the memory contents, and does not involve ownership. \texttt{AHardwareBuffer_lock} gives the host exclusive access to the buffer, is temporary, and allows for reformatting copy-in/copy-out. Implementations
are not required to support host-visible memory types for imported Android hardware buffers or resources backed by them. If a host-visible memory type is supported and used, the memory can be mapped in Vulkan, but doing so follows Vulkan semantics: it is just a raw view of the data and does not imply ownership (this means implementations must not internally call `AHardwareBuffer_lock` to implement `vkMapMemory`, or assume the application has done so). Implementations are not required to support linear-tiled images backed by Android hardware buffers, even if the `AHardwareBuffer` has CPU usage. There is no reliable way to allocate memory in Vulkan that can be exported to a `AHardwareBuffer` with CPU usage.

5) Android may add new `AHardwareBuffer` formats and usage flags over time. Can reference to them be added to this extension, or do they need a new extension?

**RESOLVED**: This extension can document the interaction between the new AHB formats/usages and existing Vulkan features. No new Vulkan features or implementation requirements can be added. The extension version number will be incremented when this additional documentation is added, but the version number does not indicate that an implementation supports Vulkan memory or resources that map to the new `AHardwareBuffer` features: support for that must be queried with `vkGetPhysicalDeviceImageFormatProperties2` or is implied by successfully allocating a `AHardwareBuffer` outside of Vulkan that uses the new feature and has a GPU usage flag.

In essence, these are new features added to a new Android API level, rather than new Vulkan features. The extension will only document how existing Vulkan features map to that new Android feature.

**Version History**

- Revision 3, 2019-08-27 (Jon Leech)
  - Update revision history to correspond to XML version number
- Revision 2, 2018-04-09 (Petr Kraus)
  - Markup fixes and remove incorrect Draft status
- Revision 1, 2018-03-04 (Jesse Hall)
  - Initial version

**VK_FUCHSIA_external_memory**

**Name String**

`VK_FUCHSIA_external_memory`

**Extension Type**

Device extension

**Registered Extension Number**

365

**Revision**

1
Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_external_memory_capabilities
• Requires VK_KHR_external_memory

Contact

• John Rosasco @rosasco

Other Extension Metadata

Last Modified Date
2021-03-01

IP Status
No known IP claims.

Contributors

• Craig Stout, Google
• John Bauman, Google
• John Rosasco, Google

Description

Vulkan apps may wish to export or import device memory handles to or from other logical devices, instances or APIs.

This memory sharing can eliminate copies of memory buffers when different subsystems need to interoperate on them. Sharing memory buffers may also facilitate a better distribution of processing workload for more complex memory manipulation pipelines.

New Commands

• vkGetMemoryZirconHandleFUCHSIA
• vkGetMemoryZirconHandlePropertiesFUCHSIA

New Structures

• VkMemoryGetZirconHandleInfoFUCHSIA
• VkMemoryZirconHandlePropertiesFUCHSIA
• Extending VkMemoryAllocateInfo:
  ◦ VkImportMemoryZirconHandleInfoFUCHSIA

New Enum Constants

• VK_FUCHSIA_EXTERNAL_MEMORY_EXTENSION_NAME
• **VK_FUCHSIA_EXTERNAL_MEMORY_SPEC_VERSION**

  Extending `VkExternalMemoryHandleTypeFlagBits`:
  
  ◦ `VK_EXTERNAL_MEMORY_HANDLE_TYPE_ZIRCON_VMO_BIT_FUCHSIA`

  Extending `VkStructureType`:
  
  ◦ `VK_STRUCTURE_TYPE_IMPORT_MEMORY_ZIRCON_HANDLE_INFO_FUCHSIA`
  ◦ `VK_STRUCTURE_TYPE_MEMORY_GET_ZIRCON_HANDLE_INFO_FUCHSIA`
  ◦ `VK_STRUCTURE_TYPE_MEMORY_ZIRCON_HANDLE_PROPERTIES_FUCHSIA`

**Issues**

See `VK_KHR_external_memory` issues list for further information.

**Version History**

• Revision 1, 2021-03-01 (John Rosasco)
  
  ◦ Initial draft

**VK_FUCHSIA_external_semaphore**

**Name String**

`VK_FUCHSIA_external_semaphore`

**Extension Type**

Device extension

**Registered Extension Number**

366

**Revision**

1

**Extension and Version Dependencies**

• Requires Vulkan 1.0

  • Requires `VK_KHR_external_semaphore_capabilities`

  • Requires `VK_KHR_external_semaphore`

**Contact**

• John Rosasco [rosasco](mailto:rosasco)

**Other Extension Metadata**

**Last Modified Date**

2021-03-08
**IP Status**

No known IP claims.

**Contributors**

- Craig Stout, Google
- John Bauman, Google
- John Rosasco, Google

**Description**

An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to export semaphore payload to and import semaphore payload from Zircon event handles.

**New Commands**

- `vkGetSemaphoreZirconHandleFUCHSIA`
- `vkImportSemaphoreZirconHandleFUCHSIA`

**New Structures**

- `VkImportSemaphoreZirconHandleInfoFUCHSIA`
- `VkSemaphoreGetZirconHandleInfoFUCHSIA`

**New Enum Constants**

- `VK_FUCHSIA_EXTERNAL_SEMAPHORE_EXTENSION_NAME`
- `VK_FUCHSIA_EXTERNAL_SEMAPHORE_SPEC_VERSION`
- Extending `VkExternalSemaphoreHandleTypeFlagBits`:
  - `VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_ZIRCON_EVENT_BIT_FUCHSIA`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_IMPORT_SEMAPHORE_ZIRCON_HANDLE_INFO_FUCHSIA`
  - `VK_STRUCTURE_TYPE_SEMAPHORE_GET_ZIRCON_HANDLE_INFO_FUCHSIA`

**Issues**

1) Does the application need to close the Zircon event handle returned by `vkGetSemaphoreZirconHandleFUCHSIA`?

**Resolved:** Yes, unless it is passed back in to a driver instance to import the semaphore. A successful get call transfers ownership of the Zircon event handle to the application, and a successful import transfers it back to the driver. Destroying the original semaphore object will not close the Zircon event handle nor remove its reference to the underlying semaphore resource associated with it.
Version History

- Revision 1, 2021-03-08 (John Rosasco)
  - Initial revision

VK_FUCHSIA_imagepipe_surface

Name String

VK_FUCHSIA_imagepipe_surface

Extension Type

Instance extension

Registered Extension Number

215

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_surface

Contact

- Craig Stout @dotstout

Other Extension Metadata

Last Modified Date

2018-07-27

IP Status

No known IP claims.

Contributors

- Craig Stout, Google
- Ian Elliott, Google
- Jesse Hall, Google

Description

The VK_FUCHSIA_imagepipe_surface extension is an instance extension. It provides a mechanism to create a VkSurfaceKHR object (defined by the VK_KHR_surface extension) that refers to a Fuchsia imagePipeHandle.
New Commands

- vkCreateImagePipeSurfaceFUCHSIA

New Structures

- VkImagePipeSurfaceCreateInfoFUCHSIA

New Bitmasks

- VkImagePipeSurfaceCreateFlagsFUCHSIA

New Enum Constants

- VK_FUCHSIA_IMAGEPIPE_SURFACE_EXTENSION_NAME
- VK_FUCHSIA_IMAGEPIPE_SURFACE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMAGEPIPE_SURFACE_CREATE_INFO_FUCHSIA

Version History

- Revision 1, 2018-07-27 (Craig Stout)
  - Initial draft.

VK_GGP_frame_token

Name String

- VK_GGP_frame_token

Extension Type

- Device extension

Registered Extension Number

- 192

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_swapchain
- Requires VK_GGP_stream_descriptor_surface

Contact

- Jean-Francois Roy @jfroy
Other Extension Metadata

Last Modified Date
2019-01-28

IP Status
No known IP claims.

Contributors
• Jean-Francois Roy, Google
• Richard O'Grady, Google

Description
This extension allows an application that uses the VK_KHR_swapchain extension in combination with a Google Games Platform surface provided by the VK_GGP_stream_descriptor_surface extension to associate a Google Games Platform frame token with a present operation.

New Structures
• Extending VkPresentInfoKHR:
  ◦ VkPresentFrameTokenGGP

New Enum Constants
• VK_GGP_FRAME_TOKEN_EXTENSION_NAME
• VK_GGP_FRAME_TOKEN_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_PRESENT_FRAME_TOKEN_GGP

Version History
• Revision 1, 2018-11-26 (Jean-Francois Roy)
  ◦ Initial revision.

VK_GGP_stream_descriptor_surface

Name String
VK_GGP_stream_descriptor_surface

Extension Type
Instance extension

Registered Extension Number
50
The **VK_GGP_stream_descriptor_surface** extension is an instance extension. It provides a mechanism to create a **VkSurfaceKHR** object (defined by the **VK_KHR_surface** extension) that refers to a Google Games Platform **GgpStreamDescriptor**.
New Commands

• vkCreateStreamDescriptorSurfaceGGP

New Structures

• VkStreamDescriptorSurfaceCreateInfoGGP

New Bitmasks

• VkStreamDescriptorSurfaceCreateFlagsGGP

New Enum Constants

• VK_GGP_STREAM_DESCRIPTOR_SURFACE_EXTENSION_NAME
• VK_GGP_STREAM_DESCRIPTOR_SURFACE_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_STREAM_DESCRIPTOR_SURFACE_CREATE_INFO_GGP

Version History

• Revision 1, 2018-11-26 (Jean-Francois Roy)
  ◦ Initial revision.

VK_GOOGLE_decorate_string

Name String
  VK_GOOGLE_decorate_string

Extension Type
  Device extension

Registered Extension Number
  225

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0

Contact
  • Hai Nguyen chaoticbob

Other Extension Metadata
Description

The `VK_GOOGLE_decorate_string` extension allows use of the `SPV_GOOGLE_decorate_string` extension in SPIR-V shader modules.

New Enum Constants

- `VK_GOOGLE_DECORATE_STRING_EXTENSION_NAME`
- `VK_GOOGLE_DECORATE_STRING_SPEC_VERSION`

Version History

- Revision 1, 2018-07-09 (Neil Henning)
  ◦ Initial draft

`VK_GOOGLE_display_timing`

Name String

`VK_GOOGLE_display_timing`

Extension Type

- Device extension

Registered Extension Number

- 93

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_swapchain`
This device extension allows an application that uses the VK_KHR_swapchain extension to obtain information about the presentation engine's display, to obtain timing information about each present, and to schedule a present to happen no earlier than a desired time. An application can use this to minimize various visual anomalies (e.g. stuttering).

Traditional game and real-time animation applications need to correctly position their geometry for when the presentable image will be presented to the user. To accomplish this, applications need various timing information about the presentation engine’s display. They need to know when presentable images were actually presented, and when they could have been presented. Applications also need to tell the presentation engine to display an image no sooner than a given time. This allows the application to avoid stuttering, so the animation looks smooth to the user.

This extension treats variable-refresh-rate (VRR) displays as if they are fixed-refresh-rate (FRR) displays.

**New Commands**

- vkGetPastPresentationTimingGOOGLE
- vkGetRefreshCycleDurationGOOGLE

**New Structures**

- VkPastPresentationTimingGOOGLE
- VkPresentTimeGOOGLE
- VkRefreshCycleDurationGOOGLE
- Extending VkPresentInfoKHR:
  - VkPresentTimesInfoGOOGLE
New Enum Constants

- `VK_GOOGLE_DISPLAY_TIMING_EXTENSION_NAME`
- `VK_GOOGLE_DISPLAY_TIMING_SPEC_VERSION`

Extending `VkStructureType`:
- `VK_STRUCTURE_TYPE_PRESENT_TIMES_INFO_GOOGLE`

Examples

Note
The example code for this extension (like the `VK_KHR_surface` and `VK_GOOGLE_display_timing` extensions) is contained in the cube demo that is shipped with the official Khronos SDK, and is being kept up-to-date in that location (see: https://github.com/KhronosGroup/Vulkan-Tools/blob/master/cube/cube.c).

Version History

- Revision 1, 2017-02-14 (Ian Elliott)
  - Internal revisions

`VK_GOOGLE_hlsl_functionality1`

Name String
`VK_GOOGLE_hlsl_functionality1`

Extension Type
Device extension

Registered Extension Number
224

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0

Contact
- Hai Nguyen

Other Extension Metadata

Last Modified Date
2018-07-09
IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_GOOGLE_hlsl_functionality1

Contributors
• Hai Nguyen, Google
• Neil Henning, AMD

Description
The VK_GOOGLE_hlsl_functionality1 extension allows use of the SPV_GOOGLE_hlsl_functionality1 extension in SPIR-V shader modules.

New Enum Constants
• VK_GOOGLE_HLSL_FUNCTIONALITY1_EXTENSION_NAME
• VK_GOOGLE_HLSL_FUNCTIONALITY1_SPEC_VERSION

Version History
• Revision 1, 2018-07-09 (Neil Henning)
  ◦ Initial draft

VK_GOOGLE_user_type

Name String
VK_GOOGLE_user_type

Extension Type
Device extension

Registered Extension Number
290

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Kaye Mason chaleur
Other Extension Metadata

Last Modified Date
2019-07-09

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_GOOGLE_user_type

Contributors
• Kaye Mason, Google
• Hai Nguyen, Google

Description
The VK_GOOGLE_user_type extension allows use of the SPV_GOOGLE_user_type extension in SPIR-V shader modules.

New Enum Constants
• VK_GOOGLE_USER_TYPE_EXTENSION_NAME
• VK_GOOGLE_USER_TYPE_SPEC_VERSION

Version History
• Revision 1, 2019-09-07 (Kaye Mason)
  ◦ Initial draft

VK_IMG_filter_cubic

Name String
VK_IMG_filter_cubic

Extension Type
Device extension

Registered Extension Number
16

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
Description

VK_IMG_filter_cubic adds an additional, high quality cubic filtering mode to Vulkan, using a Catmull-Rom bicubic filter. Performing this kind of filtering can be done in a shader by using 16 samples and a number of instructions, but this can be inefficient. The cubic filter mode exposes an optimized high quality texture sampling using fixed texture sampling functionality.

New Enum Constants

- VK_IMG_FILTER_CUBIC_EXTENSION_NAME
- VK_IMG_FILTER_CUBIC_SPEC_VERSION
- Extending VkFilter:
  - VK_FILTER_CUBIC_IMG
- Extending VkFormatFeatureFlagBits:
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG

Example

Creating a sampler with the new filter for both magnification and minification

```cpp
VkSamplerCreateInfo createInfo = {
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO // sType
    // Other members set to application-desired values
};

createInfo.magFilter = VK_FILTER_CUBIC_IMG;
createInfo.minFilter = VK_FILTER_CUBIC_IMG;

VkSampler sampler;
VkResult result = vkCreateSampler(
    device,
    &createInfo,
    &sampler);
```
Version History

• Revision 1, 2016-02-23 (Tobias Hector)
  ◦ Initial version

VK_IMG_format_pvrtc

Name String

VK_IMG_format_pvrtc

Extension Type

Device extension

Registered Extension Number

55

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Stuart Smith

Other Extension Metadata

Last Modified Date

2019-09-02

IP Status

Imagination Technologies Proprietary

Contributors

• Stuart Smith, Imagination Technologies

Description

VK_IMG_format_pvrtc provides additional texture compression functionality specific to Imagination Technologies PowerVR Texture compression format (called PVRTC).

New Enum Constants

• VK_IMG_FORMAT_PVRTC_EXTENSION_NAME
• VK_IMG_FORMAT_PVRTC_SPEC_VERSION
• Extending VkFormat:
  ◦ VK_FORMAT_PVRTC1_2BPP_SRGB_BLOCK_IMG
VK_FORMAT_PVRTC1_2BPP_UNORM_BLOCK_IMG
VK_FORMAT_PVRTC1_4BPP_SRGB_BLOCK_IMG
VK_FORMAT_PVRTC1_4BPP_UNORM_BLOCK_IMG
VK_FORMAT_PVRTC2_2BPP_SRGB_BLOCK_IMG
VK_FORMAT_PVRTC2_2BPP_UNORM_BLOCK_IMG
VK_FORMAT_PVRTC2_4BPP_SRGB_BLOCK_IMG
VK_FORMAT_PVRTC2_4BPP_UNORM_BLOCK_IMG

Version History

• Revision 1, 2019-09-02 (Stuart Smith)
  ◦ Initial version

VK_INTEL_performance_query

Name String
  VK_INTEL_performance_query

Extension Type
  Device extension

Registered Extension Number
  211

Revision
  2

Extension and Version Dependencies
  • Requires Vulkan 1.0

Special Use
  • Developer tools

Contact
  • Lionel Landwerlin @llandwerlin

Other Extension Metadata

Last Modified Date
  2018-05-16

IP Status
  No known IP claims.
Contributors

- Lionel Landwerlin, Intel
- Piotr Maciejewski, Intel

Description

This extension allows an application to capture performance data to be interpreted by an external application or library.

Such a library is available at: https://github.com/intel/metrics-discovery

Performance analysis tools such as Graphics Performance Analyzers make use of this extension and the metrics-discovery library to present the data in a human-readable way.

New Object Types

- VkPerformanceConfigurationINTEL

New Commands

- vkAcquirePerformanceConfigurationINTEL
- vkCmdSetPerformanceMarkerINTEL
- vkCmdSetPerformanceOverrideINTEL
- vkCmdSetPerformanceStreamMarkerINTEL
- vkGetPerformanceParameterINTEL
- vkInitializePerformanceApiINTEL
- vkQueueSetPerformanceConfigurationINTEL
- vkReleasePerformanceConfigurationINTEL
- vkUninitializePerformanceApiINTEL

New Structures

- VkInitializePerformanceApiInfoINTEL
- VkPerformanceConfigurationAcquireInfoINTEL
- VkPerformanceMarkerInfoINTEL
- VkPerformanceOverrideInfoINTEL
- VkPerformanceStreamMarkerInfoINTEL
- VkPerformanceValueINTEL
- Extending VkQueryPoolCreateInfo:
  - VkQueryPoolPerformanceQueryCreateInfoINTEL
New Unions

- VkPerformanceValueDataINTEL

New Enums

- VkPerformanceConfigurationTypeINTEL
- VkPerformanceOverrideTypeINTEL
- VkPerformanceParameterTypeINTEL
- VkPerformanceValueTypeINTEL
- VkQueryPoolSamplingModeINTEL

New Enum Constants

- VK_INTEL_PERFORMANCE_QUERY_EXTENSION_NAME
- VK_INTEL_PERFORMANCE_QUERY_SPEC_VERSION

Extending VkObjectType:

- VK_OBJECT_TYPE_PERFORMANCE_CONFIGURATION_INTEL

Extending VkQueryType:

- VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL

Extending VkStructureType:

- VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL
- VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL
- VK_STRUCTURE_TYPE_PERFORMANCE_MARKER_INFO_INTEL
- VK_STRUCTURE_TYPE_PERFORMANCE_OVERRIDE_INFO_INTEL
- VK_STRUCTURE_TYPE_PERFORMANCE_STREAM_MARKER_INFO_INTEL
- VK_STRUCTURE_TYPE_QUERY_POOL_PERFORMANCE_QUERY_CREATE_INFO_INTEL

Example Code

```c
// A previously created device
VkDevice device;

// A queue derived from the device
VkQueue queue;

VkInitializePerformanceApiInfoINTEL performanceApiInfoIntel = {
    VK_STRUCTURE_TYPE_INITIALIZE_PERFORMANCE_API_INFO_INTEL,
    NULL,
    NULL
};
```
vkInitializePerformanceApiINTEL(
    device,
    &performanceApiInfoIntel);

VkQueryPoolPerformanceQueryCreateInfoINTEL queryPoolIntel = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO_INTEL,
    NULL,
    VK_QUERY_POOL_SAMPLING_MODE_MANUAL_INTEL,
};

VkQueryPoolCreateInfo queryPoolCreateInfo = {
    VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO,
    &queryPoolIntel,
    0,
    VK_QUERY_TYPE_PERFORMANCE_QUERY_INTEL,
    1,
    0
};

VkQueryPool queryPool;

VkResult result = vkCreateQueryPool(
    device,
    &queryPoolCreateInfo,
    NULL,
    &queryPool);

assert(VK_SUCCESS == result);

// A command buffer we want to record counters on
VkCommandBuffer commandBuffer;

VkCommandBufferBeginInfo commandBufferBeginInfo = {
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
    NULL,
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT,
    NULL
};

result = vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);

assert(VK_SUCCESS == result);

vkCmdResetQueryPool(
    commandBuffer,
    queryPool,
    0,
    1);

vkCmdBeginQuery(
    commandBuffer,
queryPool,
0,
0);

// Perform the commands you want to get performance information on
// ...

// Perform a barrier to ensure all previous commands were complete before
// ending the query
vkCmdPipelineBarrier(commandBuffer,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
0,
0,
NULL,
0,
NULL,
0,
NULL);

vkCmdEndQuery(
    commandBuffer,
    queryPool,
    0);

result = vkEndCommandBuffer(commandBuffer);

assert(VK_SUCCESS == result);

VkPerformanceConfigurationAcquireInfoINTEL performanceConfigurationAcquireInfo = {
    VK_STRUCTURE_TYPE_PERFORMANCE_CONFIGURATION_ACQUIRE_INFO_INTEL,
    NULL,
    VK_PERFORMANCE_CONFIGURATION_TYPE_COMMAND_QUEUE_METRICS_DISCOVERY_ACTIVATED_INTEL
};

VkPerformanceConfigurationINTEL performanceConfigurationIntel;

result = vkAcquirePerformanceConfigurationINTEL(
    device,
    &performanceConfigurationAcquireInfo,
    &performanceConfigurationIntel);

vkQueueSetPerformanceConfigurationINTEL(queue, performanceConfigurationIntel);

assert(VK_SUCCESS == result);

// Submit the command buffer and wait for its completion
// ...

result = vkReleasePerformanceConfigurationINTEL(
    device,
performanceConfigurationIntel);

assert(VK_SUCCESS == result);

// Get the report size from metrics-discovery's QueryReportSize

result = vkGetQueryPoolResults(
    device,
    queryPool,
    0, 1, QueryReportSize,
    data, QueryReportSize, 0);

assert(VK_SUCCESS == result);

// The data can then be passed back to metrics-discovery from which
// human readable values can be queried.

Version History

• Revision 2, 2020-03-06 (Lionel Landwerlin)
  ◦ Rename VkQueryPoolCreateInfoINTEL in VkQueryPoolPerformanceQueryCreateInfoINTEL

• Revision 1, 2018-05-16 (Lionel Landwerlin)
  ◦ Initial revision

VK_INTEL_shader_integer_functions2

Name String

VK_INTEL_shader_integer_functions2

Extension Type

Device extension

Registered Extension Number

210

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0

• Requires VK_KHR_get_physical_device_properties2

Contact

• Ian Romanick 🌐ianromanick
Description

This extension adds support for several new integer instructions in SPIR-V for use in graphics shaders. Many of these instructions have pre-existing counterparts in the Kernel environment.

The added integer functions are defined by the `SPV_INTEL_shader_integer_functions` SPIR-V extension and can be used with the `GL_INTEL_shader_integer_functions2` GLSL extension.

New Structures

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceShaderIntegerFunctions2FeaturesINTEL`

New Enum Constants

- `VK_INTEL_SHADER_INTEGER_FUNCTIONS_2_EXTENSION_NAME`
- `VK_INTEL_SHADER_INTEGER_FUNCTIONS_2_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_INTEGER_FUNCTIONS_2_FEATURES_INTEL`

New SPIR-V Capabilities

- `IntegerFunctions2INTEL`

Version History

- Revision 1, 2019-04-30 (Ian Romanick)
  - Initial draft

VK_NN_vi_surface

Name String

`VK_NN_vi_surface`
Extension Type
  Instance extension

Registered Extension Number
  63

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_surface

Contact
  • Mathias Heyer mheyer

Other Extension Metadata

Last Modified Date
  2016-12-02

IP Status
  No known IP claims.

Contributors
  • Mathias Heyer, NVIDIA
  • Michael Chock, NVIDIA
  • Yasuhiro Yoshioka, Nintendo
  • Daniel Koch, NVIDIA

Description

The VK_NN_vi_surface extension is an instance extension. It provides a mechanism to create a VkSurfaceKHR object (defined by the VK_KHR_surface extension) associated with an nn::vi::Layer.

New Commands
  • vkCreateViSurfaceNN

New Structures
  • VkViSurfaceCreateInfoNN

New Bitmasks
  • VkViSurfaceCreateFlagsNN
New Enum Constants

- VK_NN_VI_SURFACE_EXTENSION_NAME
- VK_NN_VI_SURFACE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_VI_SURFACE_CREATE_INFO_NN

Issues

1) Does VI need a way to query for compatibility between a particular physical device (and queue family?) and a specific VI display?

**RESOLVED**: No. It is currently always assumed that the device and display will always be compatible.

2) VkViSurfaceCreateInfoNN::pWindow is intended to store an nn::vi::NativeWindowHandle, but its declared type is a bare void* to store the window handle. Why the discrepancy?

**RESOLVED**: It is for C compatibility. The definition for the VI native window handle type is defined inside the nn::vi C++ namespace. This prevents its use in C source files. nn::vi::NativeWindowHandle is always defined to be void*, so this extension uses void* to match.

Version History

- Revision 1, 2016-12-2 (Michael Chock)
  - Initial draft.

**VK_NV_acquire_winrt_display**

Name String

- VK_NV_acquire_winrt_display

Extension Type

- Device extension

Registered Extension Number

- 346

Revision

- 1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_EXT_direct_mode_display

Contact

- Jeff Juliano jjuliano
Other Extension Metadata

Last Modified Date
2020-09-29

IP Status
No known IP claims.

Contributors
• Jeff Juliano, NVIDIA

Description
This extension allows an application to take exclusive control of a display on Windows 10 provided that the display is not already controlled by a compositor. Examples of compositors include the Windows desktop compositor, other applications using this Vulkan extension, and applications that “Acquire” a “DisplayTarget” using a “WinRT” command such as “winrt::Windows::Devices::Display::Core::DisplayManager.TryAcquireTarget()”.

When control is acquired the application has exclusive access to the display until control is released or the application terminates. An application’s attempt to acquire is denied if a different application has already acquired the display.

New Commands
• vkAcquireWinrtDisplayNV
• vkGetWinrtDisplayNV

New Enum Constants
• VK_NV_ACQUIRE_WINRT_DISPLAY_EXTENSION_NAME
• VK_NV_ACQUIRE_WINRT_DISPLAY_SPEC_VERSION

Issues
1) What should the platform substring be for this extension:

RESOLVED: The platform substring is “Winrt”.

The substring “Winrt” matches the fact that the OS API exposing the acquire and release functionality is called “WinRT”.

The substring “Win32” is wrong because the related “WinRT” API is explicitly not a “Win32” API. “WinRT” is a competing API family to the “Win32” API family.

The substring “Windows” is suboptimal because there could be more than one relevant API on the Windows platform. There is preference to use the more-specific substring “Winrt”.

2) Should vkAcquireWinrtDisplayNV take a winRT DisplayTarget, or a Vulkan display handle as...
RESOLVED: A Vulkan display handle. This matches the design of vkAcquireXlibDisplayEXT.

3) Should the acquire command be platform-independent named “vkAcquireDisplayNV”, or platform-specific named “vkAcquireWinrtDisplayNV”?

RESOLVED: Add a platform-specific command.

The inputs to the Acquire command are all Vulkan types. None are WinRT types. This opens the possibility of the winrt extension defining a platform-independent acquire command.

The X11 acquire command does need to accept a platform-specific parameter. This could be handled by adding to a platform-independent acquire command a params struct to which platform-dependent types can be chained by pNext pointer.

The prevailing opinion is that it would be odd to create a second platform-independent function that is used on the Windows 10 platform, but that is not used for the X11 platform. Since a Windows 10 platform-specific command is needed anyway for converting between vkDisplayKHR and platform-native handles, opinion was to create a platform-specific acquire function.

4) Should the vkGetWinrtDisplayNV parameter identifying a display be named “deviceRelativeId” or “adapterRelativeId”?

RESOLVED: The WinRT name is “AdapterRelativeId”. The name “adapter” is the Windows analog to a Vulkan “physical device”. Vulkan already has precedent to use the name deviceUID for the concept that Windows APIs call “AdapterLuid”. Keeping form with this precedent, the name “deviceRelativeId” is chosen.

5) Does vkAcquireWinrtDisplayNV cause the Windows desktop compositor to release a display?

RESOLVED: No. vkAcquireWinrtDisplayNV does not itself cause the Windows desktop compositor to release a display. This action must be performed outside of Vulkan.

Beginning with Windows 10 version 2004 it is possible to cause the Windows desktop compositor to release a display by using the “Advanced display settings” sub-page of the “Display settings” control panel. See https://docs.microsoft.com/en-us/windows-hardware/drivers/display/specialized-monitors

6) Where can one find additional information about custom compositors for Windows 10?

RESOLVED: Relevant references are as follows.

According to Microsoft’s documentation on “building a custom compositor”, the ability to write a custom compositor is not a replacement for a fullscreen desktop window. The feature is for writing compositor apps that drive specialized hardware.

Only certain editions of Windows 10 support custom compositors, "documented here”. The product type can be queried from Windows 10. See https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getproductinfo
Virtual Reality (VR) applications often involve a post-processing step to apply a “barrel” distortion to the rendered image to correct the “pincushion” distortion introduced by the optics in a VR device. The barrel distorted image has lower resolution along the edges compared to the center. Since the original image is rendered at high resolution, which is uniform across the complete image, a lot of pixels towards the edges do not make it to the final post-processed image.

This extension provides a mechanism to render VR scenes at a non-uniform resolution, in particular a resolution that falls linearly from the center towards the edges. This is achieved by scaling the w coordinate of the vertices in the clip space before perspective divide. The clip space w coordinate of the vertices can be offset as a function of x and y coordinates as follows:

\[ w' = w + Ax + By \]
In the intended use case for viewport position scaling, an application should use a set of four viewports, one for each of the four quadrants of a Cartesian coordinate system. Each viewport is set to the dimension of the image, but is scissored to the quadrant it represents. The application should specify A and B coefficients of the w-scaling equation above, that have the same value, but different signs, for each of the viewports. The signs of A and B should match the signs of x and y for the quadrant that they represent such that the value of w' will always be greater than or equal to the original w value for the entire image. Since the offset to w, (Ax + By), is always positive, and increases with the absolute values of x and y, the effective resolution will fall off linearly from the center of the image to its edges.

**New Commands**

- `vkCmdSetViewportWScalingNV`

**New Structures**

- `VkViewportWScalingNV`
- Extending `VkPipelineViewportStateCreateInfo`:
  - `VkPipelineViewportWScalingStateCreateInfoNV`

**New Enum Constants**

- `VK_NV_CLIP_SPACE_W_SCALING_EXTENSION_NAME`
- `VK_NV_CLIP_SPACE_W_SCALING_SPEC_VERSION`
- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_VIEWPORT_W_SCALING_NV`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV`

**Issues**

1) Is the pipeline struct name too long?

RESOLVED: It fits with the naming convention.

2) Separate W scaling section or fold into coordinate transformations?

RESOLVED: Leaving it as its own section for now.

**Examples**
VkViewport viewports[4];
VkRect2D scissors[4];
VkViewportWScalingNV scalings[4];

for (int i = 0; i < 4; i++) {
    int x = (i & 2) ? 0 : currentWindowWidth / 2;
    int y = (i & 1) ? 0 : currentWindowHeight / 2;

    viewports[i].x = 0;
    viewports[i].y = 0;
    viewports[i].width = currentWindowWidth;
    viewports[i].height = currentWindowHeight;
    viewports[i].minDepth = 0.0f;
    viewports[i].maxDepth = 1.0f;

    scissors[i].offset.x = x;
    scissors[i].offset.y = y;
    scissors[i].extent.width = currentWindowWidth / 2;
    scissors[i].extent.height = currentWindowHeight / 2;

    const float factor = 0.15;
    scalings[i].xcoeff = ((i & 2) ? -1.0 : 1.0) * factor;
    scalings[i].ycoeff = ((i & 1) ? -1.0 : 1.0) * factor;
}

VkPipelineViewportWScalingStateCreateInfoNV vpWScalingStateInfo = {
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_W_SCALING_STATE_CREATE_INFO_NV }

vpWScalingStateInfo.viewportWScalingEnable = VK_TRUE;
vpWScalingStateInfo.viewportCount = 4;
vpWScalingStateInfo.pViewportWScalings = &scalings[0];

VkPipelineViewportStateCreateInfo vpStateInfo = {
    VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO }

vpStateInfo.viewportCount = 4;
vpStateInfo.pViewports = &viewports[0];
vpStateInfo.scissorCount = 4;
vpStateInfo.pScissors = &scissors[0];
vpStateInfo.pNext = &vpWScalingStateInfo;

Example shader to read from a w-scaled texture:
// Vertex Shader
// Draw a triangle that covers the whole screen
const vec4 positions[3] = vec4[3](vec4(-1, -1, 0, 1),
    vec4( 3, -1, 0, 1),
    vec4(-1,  3, 0, 1));

out vec2 uv;
void main()
{
    vec4 pos = positions[ gl_VertexID ];
    gl_Position = pos;
    uv = pos.xy;
}

// Fragment Shader
uniform sampler2D tex;
uniform float xcoeff;
uniform float ycoeff;
out vec4 Color;
in vec2 uv;
void main()
{
    // Handle uv as if upper right quadrant
    vec2 uvabs = abs(uv);

    // unscale: transform w-scaled image into an unscaled image
    // scale: transform unscaled image into a w-scaled image
    float unscale = 1.0 / (1 + xcoeff * uvabs.x + xcoeff * uvabs.y);
    //float scale = 1.0 / (1 - xcoeff * uvabs.x - xcoeff * uvabs.y);

    vec2 P = vec2(unscale * uvabs.x, unscale * uvabs.y);

    // Go back to the right quadrant
    P *= sign(uv);

    Color = texture(tex, P * 0.5 + 0.5);
}

**Version History**

- Revision 1, 2017-02-15 (Eric Werness)
  - Internal revisions

**VK_NV_compute_shader_derivatives**

**Name String**

VK_NV_compute_shader_derivatives
Extension Type
Device extension

Registered Extension Number
202

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact
• Pat Brown nvpbrown

Other Extension Metadata

Last Modified Date
2018-07-19

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPIR-V extension SPV_NV_compute_shader_derivatives
• This extension provides API support for GL_NV_compute_shader_derivatives

Contributors
• Pat Brown, NVIDIA

Description
This extension adds Vulkan support for the SPV_NV_compute_shader_derivatives SPIR-V extension.

The SPIR-V extension provides two new execution modes, both of which allow compute shaders to use built-ins that evaluate compute derivatives explicitly or implicitly. Derivatives will be computed via differencing over a 2x2 group of shader invocations. The DerivativeGroupQuadsNV execution mode assembles shader invocations into 2x2 groups, where each group has x and y coordinates of the local invocation ID of the form (2m+{0,1}, 2n+{0,1}). The DerivativeGroupLinearNV execution mode assembles shader invocations into 2x2 groups, where each group has local invocation index values of the form 4m+{0,1,2,3}.

New Structures
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceComputeShaderDerivativesFeaturesNV
New Enum Constants

- VK_NV_COMPUTE_SHADER_DERIVATIVES_EXTENSION_NAME
- VK_NV_COMPUTE_SHADER_DERIVATIVES_SPEC_VERSION

Extending VkStructureType:
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COMPUTE_SHADER_DERIVATIVES_FEATURES_NV

New SPIR-V Capability

- ComputeDerivativeGroupQuadsNV
- ComputeDerivativeGroupLinearNV

Issues

(1) Should we specify that the groups of four shader invocations used for derivatives in a compute shader are the same groups of four invocations that form a “quad” in shader subgroups?

RESOLVED: Yes.

Examples

None.

Version History

- Revision 1, 2018-07-19 (Pat Brown)
  - Initial draft

VK_NV_cooperative_matrix

Name String

VK_NV_cooperative_matrix

Extension Type

Device extension

Registered Extension Number

250

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2
Contact
• Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

Other Extension Metadata

Last Modified Date
2019-02-05

Interactions and External Dependencies
• This extension requires SPV_NV_cooperative_matrix
• This extension provides API support for GL_NV_cooperative_matrix

Contributors
• Jeff Bolz, NVIDIA
• Markus Tavenrath, NVIDIA
• Daniel Koch, NVIDIA

Description

This extension adds support for using cooperative matrix types in SPIR-V. Cooperative matrix types are medium-sized matrices that are primarily supported in compute shaders, where the storage for the matrix is spread across all invocations in some scope (usually a subgroup) and those invocations cooperate to efficiently perform matrix multiplies.

Cooperative matrix types are defined by the SPV_NV_cooperative_matrix SPIR-V extension and can be used with the GL_NV_cooperative_matrix GLSL extension.

This extension includes support for enumerating the matrix types and dimensions that are supported by the implementation.

New Commands
• vkGetPhysicalDeviceCooperativeMatrixPropertiesNV

New Structures
• VkCooperativeMatrixPropertiesNV
• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceCooperativeMatrixFeaturesNV
• Extending VkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceCooperativeMatrixPropertiesNV

New Enums
• VkComponentTypeNV
New Enum Constants

- VK_NV_COOPERATIVE_MATRIX_EXTENSION_NAME
- VK_NV_COOPERATIVE_MATRIX_SPEC_VERSION

Extending VkStructureType:
- VK_STRUCTURE_TYPE_COOPERATIVE_MATRIX_PROPERTIES_NV
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_COOPERATIVE_MATRIX_FEATURES_NV
- VK_STRUCTURE_TYPE_PHYSICALDEVICE_COOPERATIVE_MATRIX_PROPERTIES_NV

New SPIR-V Capabilities

- CooperativeMatrixNV

Issues

(1) What matrix properties will be supported in practice?

**RESOLVED**: In NVIDIA’s initial implementation, we will support:

- AType = BType = fp16 CType = DType = fp16 MxNxK = 16x8x16 scope = Subgroup
- AType = BType = fp16 CType = DType = fp16 MxNxK = 16x8x8 scope = Subgroup
- AType = BType = fp16 CType = DType = fp32 MxNxK = 16x8x16 scope = Subgroup
- AType = BType = fp16 CType = DType = fp32 MxNxK = 16x8x8 scope = Subgroup

Version History

- Revision 1, 2019-02-05 (Jeff Bolz)
- Internal revisions

**VK_NV_corner_sampled_image**

Name String

VK_NV_corner_sampled_image

Extension Type

Device extension

Registered Extension Number

51

Revision

2
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Contact

- Daniel Koch

Other Extension Metadata

Last Modified Date

2018-08-13

Contributors

- Jeff Bolz, NVIDIA
- Pat Brown, NVIDIA
- Chris Lentini, NVIDIA

Description

This extension adds support for a new image organization, which this extension refers to as “corner-sampled” images. A corner-sampled image differs from a conventional image in the following ways:

- Texels are centered on integer coordinates. See [Unnormalized Texel Coordinate Operations](#).
- Normalized coordinates are scaled using $\text{coord} \times (\text{dim} - 1)$ rather than $\text{coord} \times \text{dim}$, where dim is the size of one dimension of the image. See [normalized texel coordinate transform](#).
- Partial derivatives are scaled using $\text{coord} \times (\text{dim} - 1)$ rather than $\text{coord} \times \text{dim}$. See [Scale Factor Operation](#).
- Calculation of the next higher lod size goes according to $\lceil \text{dim} / 2 \rceil$ rather than $\lfloor \text{dim} / 2 \rfloor$. See [Image Miplevel Sizing](#).
- The minimum level size is 2x2 for 2D images and 2x2x2 for 3D images. See [Image Miplevel Sizing](#).

This image organization is designed to facilitate a system like Ptex with separate textures for each face of a subdivision or polygon mesh. Placing sample locations at pixel corners allows applications to maintain continuity between adjacent patches by duplicating values along shared edges. Additionally, using the modified mipmapping logic along with texture dimensions of the form $2^n+1$ allows continuity across shared edges even if the adjacent patches use different level-of-detail values.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceCornerSampledImageFeaturesNV`
New Enum Constants

- VK_NV_CORNER_SAMPLED_IMAGE_EXTENSION_NAME
- VK_NV_CORNER_SAMPLED_IMAGE_SPEC_VERSION
- Extending VkImageCreateFlagBits:
  - VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_CORNER_SAMPLED_IMAGE_FEATURES_NV

Issues

1. What should this extension be named?

   DISCUSSION: While naming this extension, we chose the most distinctive aspect of the image organization and referred to such images as “corner-sampled images”. As a result, we decided to name the extension NV_corner_sampled_image.

2. Do we need a format feature flag so formats can advertise if they support corner-sampling?

   DISCUSSION: Currently NVIDIA supports this for all 2D and 3D formats, but not for cubemaps or depth-stencil formats. A format feature might be useful if other vendors would only support this on some formats.

3. Do integer texel coordinates have a different range for corner-sampled images?

   RESOLVED: No, these are unchanged.

4. Do unnormalized sampler coordinates work with corner-sampled images? Are there any functional differences?

   RESOLVED: Yes. Unnormalized coordinates are treated as already scaled for corner-sample usage.

5. Should we have a diagram in the “Image Operations” chapter demonstrating different texel sampling locations?

   UNRESOLVED: Probably, but later.

Version History

- Revision 1, 2018-08-14 (Daniel Koch)
  - Internal revisions
- Revision 2, 2018-08-14 (Daniel Koch)
  - ???
VK_NV_coverage_reduction_mode

Name String
   VK_NV_coverage_reduction_mode

Extension Type
   Device extension

Registered Extension Number
   251

Revision
   1

Extension and Version Dependencies
   • Requires Vulkan 1.0
   • Requires VK_NV_framebuffer_mixed_samples

Contact
   • Kedarnath Thangudu @kthangudu

Other Extension Metadata

Last Modified Date
   2019-01-29

Contributors
   • Kedarnath Thangudu, NVIDIA
   • Jeff Bolz, NVIDIA

Description

When using a framebuffer with mixed samples, a per-fragment coverage reduction operation is performed which generates color sample coverage from the pixel coverage. This extension defines the following modes to control how this reduction is performed.

• Merge: When there are more samples in the pixel coverage than color samples, there is an implementation dependent association of each pixel coverage sample to a color sample. In the merge mode, the color sample coverage is computed such that only if any associated sample in the pixel coverage is covered, the color sample is covered. This is the default mode.

• Truncate: When there are more raster samples (N) than color samples (M), there is one to one association of the first M raster samples to the M color samples; other raster samples are ignored.

When the number of raster samples is equal to the color samples, there is a one to one mapping between them in either of the above modes.

The new command `vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV` can...
be used to query the various raster, color, depth/stencil sample count and reduction mode combinations that are supported by the implementation. This extension would allow an implementation to support the behavior of both VK_NV_framebuffer_mixed_samples and VK_AMD_mixed_attachment_samples extensions simultaneously.

**New Commands**

- vkGetPhysicalDeviceSupportedFramebufferMixedSamplesCombinationsNV

**New Structures**

- VkFramebufferMixedSamplesCombinationNV
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceCoverageReductionModeFeaturesNV
- Extending VkPipelineMultisampleStateCreateInfo:
  - VkPipelineCoverageReductionStateCreateInfoNV

**New Enums**

- VkCoverageReductionModeNV

**New Bitmasks**

- VkPipelineCoverageReductionStateCreateFlagsNV

**New Enum Constants**

- VK_NV_COVERAGE_REDUCTION_MODE_EXTENSION_NAME
- VK_NV_COVERAGE_REDUCTION_MODE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_FRAMEBUFFER_MIXED_SAMPLES_COMBINATION_NV
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COVERAGE_REDUCTION_MODE_FEATURES_NV
  - VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_REDUCTION_STATE_CREATE_INFO_NV

**Version History**

- Revision 1, 2019-01-29 (Kedarnath Thangudu)
  - Internal revisions

**VK_NV_dedicated_allocation_image_aliasing**

**Name String**

VK_NV_dedicated_allocation_image_aliasing
Description

This extension allows applications to alias images on dedicated allocations, subject to specific restrictions: the extent and the number of layers in the image being aliased must be smaller than or equal to those of the original image for which the allocation was created, and every other image parameter must match.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceDedicatedAllocationImageAliasingFeaturesNV`

New Enum Constants

- `VK_NV_DEDICATED_ALLOCATION_IMAGE_ALIASING_EXTENSION_NAME`
- `VK_NV_DEDICATED_ALLOCATION_IMAGE_ALIASING_SPEC_VERSION`

Extending `VkStructureType`:
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEDICATED_ALLOCATION_IMAGE_ALIASING_FEATURES_NV`
Version History

- Revision 1, 2019-01-04 (Nuno Subtil)
  - Internal revisions

VK_NV_device_diagnostic_checkpoints

Name String
  VK_NV_device_diagnostic_checkpoints

Extension Type
  Device extension

Registered Extension Number
  207

Revision
  2

Extension and Version Dependencies
  - Requires Vulkan 1.0
  - Requires VK_KHR_get_physical_device_properties2

Contact
  - Nuno Subtil nsubtil

Other Extension Metadata

Last Modified Date
  2018-07-16

Contributors
  - Oleg Kuznetsov, NVIDIA
  - Alex Dunn, NVIDIA
  - Jeff Bolz, NVIDIA
  - Eric Werness, NVIDIA
  - Daniel Koch, NVIDIA

Description

This extension allows applications to insert markers in the command stream and associate them with custom data.

If a device lost error occurs, the application may then query the implementation for the last markers to cross specific implementation-defined pipeline stages, in order to narrow down which commands were executing at the time and might have caused the failure.
New Commands

• vkCmdSetCheckpointNV
• vkGetQueueCheckpointDataNV

New Structures

• VkCheckpointDataNV
• Extending VkQueueFamilyProperties2:
  ◦ VkQueueFamilyCheckpointPropertiesNV

New Enum Constants

• VK_NV_DEVICE_DIAGNOSTIC_CHECKPOINTS_EXTENSION_NAME
• VK_NV_DEVICE_DIAGNOSTIC_CHECKPOINTS_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_CHECKPOINT_DATA_NV
  ◦ VK_STRUCTURE_TYPE_QUEUE_FAMILY_CHECKPOINT_PROPERTIES_NV

Version History

• Revision 1, 2018-07-16 (Nuno Subtil)
  ◦ Internal revisions
• Revision 2, 2018-07-16 (Nuno Subtil)
  ◦ ???

VK_NV_device_diagnostics_config

Name String

  VK_NV_device_diagnostics_config

Extension Type

  Device extension

Registered Extension Number

  301

Revision

  1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2
Applications using Nvidia Nsight™ Aftermath SDK for Vulkan to integrate device crash dumps into their error reporting mechanisms, may use this extension to configure options related to device crash dump creation.

**New Structures**
- Extending `VkDeviceCreateInfo`:
  - `VkDeviceDiagnosticsConfigCreateInfoNV`
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceDiagnosticsConfigFeaturesNV`

**New Enums**
- `VkDeviceDiagnosticsConfigFlagBitsNV`

**New Bitmasks**
- `VkDeviceDiagnosticsConfigFlagsNV`

**New Enum Constants**
- `VK_NV_DEVICE_DIAGNOSTICS_CONFIG_EXTENSION_NAME`
- `VK_NV DEVICE_DIAGNOSTICS_CONFIG_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DEVICE_DIAGNOSTICS_CONFIG_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DIAGNOSTICS_CONFIG_FEATURES_NV`

**Version History**
- Revision 1, 2019-11-21 (Kedarnath Thangudu)
  - Internal revisions
VK_NV_device_generated_commands

Name String
VK_NV_device_generated_commands

Extension Type
Device extension

Registered Extension Number
278

Revision
3

Extension and Version Dependencies
• Requires Vulkan 1.1

Contact
• Christoph Kubisch pixeljetstream

Other Extension Metadata

Last Modified Date
2020-02-20

Interactions and External Dependencies
• This extension requires Vulkan 1.1
• This extension requires VK_EXT_buffer_device_address or VK_KHR_buffer_device_address or Vulkan 1.2 for the ability to bind vertex and index buffers on the device.
• This extension interacts with VK_NV_mesh_shader. If the latter extension is not supported, remove the command token to initiate mesh tasks drawing in this extension.

Contributors
• Christoph Kubisch, NVIDIA
• Pierre Boudier, NVIDIA
• Jeff Bolz, NVIDIA
• Eric Werness, NVIDIA
• Yuriy O'Donnell, Epic Games
• Baldur Karlsson, Valve
• Mathias Schott, NVIDIA
• Tyson Smith, NVIDIA
• Ingo Esser, NVIDIA
Description

This extension allows the device to generate a number of critical graphics commands for command buffers.

When rendering a large number of objects, the device can be leveraged to implement a number of critical functions, like updating matrices, or implementing occlusion culling, frustum culling, front to back sorting, etc. Implementing those on the device does not require any special extension, since an application is free to define its own data structures, and just process them using shaders.

However, if the application desires to quickly kick off the rendering of the final stream of objects, then unextended Vulkan forces the application to read back the processed stream and issue graphics command from the host. For very large scenes, the synchronization overhead and cost to generate the command buffer can become the bottleneck. This extension allows an application to generate a device side stream of state changes and commands, and convert it efficiently into a command buffer without having to read it back to the host.

Furthermore, it allows incremental changes to such command buffers by manipulating only partial sections of a command stream — for example pipeline bindings. Unextended Vulkan requires recreation of entire command buffers in such a scenario, or updates synchronized on the host.

The intended usage for this extension is for the application to:

- create VkBuffer objects and retrieve physical addresses from them via vkGetBufferDeviceAddressEXT
- create a graphics pipeline using VkGraphicsPipelineShaderGroupsCreateInfoNV for the ability to change shaders on the device.
- create a VkIndirectCommandsLayoutNV, which lists the VkIndirectCommandsTokenTypeNV it wants to dynamically execute as an atomic command sequence. This step likely involves some internal device code compilation, since the intent is for the GPU to generate the command buffer in the pipeline.
- fill the input stream buffers with the data for each of the inputs it needs. Each input is an array that will be filled with token-dependent data.
- set up a preprocess VkBuffer that uses memory according to the information retrieved via vkGetGeneratedCommandsMemoryRequirementsNV.
- optionally preprocess the generated content using vkCmdPreprocessGeneratedCommandsNV, for example on an asynchronous compute queue, or for the purpose of re-using the data in multiple executions.
- call vkCmdExecuteGeneratedCommandsNV to create and execute the actual device commands for all sequences based on the inputs provided.

For each draw in a sequence, the following can be specified:

- a different shader group
- a number of vertex buffer bindings
- a different index buffer, with an optional dynamic offset and index type
• a number of different push constants
• a flag that encodes the primitive winding

While the GPU can be faster than a CPU to generate the commands, it will not happen asynchronously to the device, therefore the primary use-case is generating “less” total work (occlusion culling, classification to use specialized shaders, etc.).

**New Object Types**

• VkIndirectCommandsLayoutNV

**New Commands**

• vkCmdBindPipelineShaderGroupNV
• vkCmdExecuteGeneratedCommandsNV
• vkCmdPreprocessGeneratedCommandsNV
• vkCreateIndirectCommandsLayoutNV
• vkDestroyIndirectCommandsLayoutNV
• vkGetGeneratedCommandsMemoryRequirementsNV

**New Structures**

• VkBindIndexBufferIndirectCommandNV
• VkBindShaderGroupIndirectCommandNV
• VkBindVertexBufferIndirectCommandNV
• VkGeneratedCommandsInfoNV
• VkGeneratedCommandsMemoryRequirementsInfoNV
• VkGraphicsShaderGroupCreateInfoNV
• VkIndirectCommandsLayoutCreateInfoNV
• VkIndirectCommandsLayoutTokenNV
• VkIndirectCommandsStreamNV
• VkSetStateFlagsIndirectCommandNV

**Extending** VkGraphicsPipelineCreateInfo:

• Extending VkGraphicsPipelineShaderGroupsCreateInfoNV

**Extending** VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:

• Extending VkPhysicalDeviceDeviceGeneratedCommandsFeaturesNV

**Extending** VkPhysicalDeviceProperties2:

• Extending VkPhysicalDeviceDeviceGeneratedCommandsPropertiesNV
New Enums

- VkIndirectCommandsLayoutUsageFlagBitsNV
- VkIndirectCommandsTokenTypeNV
- VkIndirectStateFlagBitsNV

New Bitmasks

- VkIndirectCommandsLayoutUsageFlagsNV
- VkIndirectStateFlagsNV

New Enum Constants

- VK_NV_DEVICE_GENERATED_COMMANDS_EXTENSION_NAME
- VK_NV_DEVICE_GENERATED_COMMANDS_SPEC_VERSION

- Extending VkAccessFlagBits:
  - VK_ACCESS_COMMAND_PREPROCESS_READ_BIT_NV
  - VK_ACCESS_COMMAND_PREPROCESS_WRITE_BIT_NV

- Extending VkObjectType:
  - VK_OBJECT_TYPE_INDIRECT_COMMANDS_LAYOUT_NV

- Extending VkPipelineCreateFlagBits:
  - VK_PIPELINE_CREATE_INDIRECT_BINDABLE_BIT_NV

- Extending VkPipelineStageFlagBits:
  - VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_GENERATED_COMMANDS_INFO_NV
  - VK_STRUCTURE_TYPE_GENERATED_COMMANDS_MEMORY_REQUIREMENTS_INFO_NV
  - VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_SHADER_GROUPS_CREATE_INFO_NV
  - VK_STRUCTURE_TYPE_GRAPHICS_SHADER_GROUP_CREATE_INFO_NV
  - VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_CREATE_INFO_NV
  - VK_STRUCTURE_TYPE_INDIRECT_COMMANDS_LAYOUT_TOKEN_NV
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_FEATURES_NV
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DEVICE_GENERATED_COMMANDS_PROPERTIES_NV

Issues

1) How to name this extension?

VK_NV_device_generated_commands

As usual, one of the hardest issues ;)

Appendix E: Layers & Extensions (Informative) | 2903
Alternatives: VK_gpu_commands, VK_execute_commands, VK_device_commands, VK_device_execute_commands, VK_device_execute, VK_device_created_commands, VK_device_recorded_commands, VK_device_generated_commands, VK_indirect_generated_commands

2) Should we use a serial stateful token stream or stateless sequence descriptions?

Similarly to VkPipeline, fixed layouts have the most likelihood to be cross-vendor adoptable. They also benefit from being processable in parallel. This is a different design choice compared to the serial command stream generated through GL_NV_command_list.

3) How to name a sequence description?

VkIndirectCommandsLayout as in the NVX extension predecessor.
Alternative: VkGeneratedCommandsLayout

4) Do we want to provide indirectCommands inputs with layout or at indirectCommands time?

Separate layout from data as Vulkan does. Provide full flexibility for indirectCommands.

5) Should the input be provided as SoA or AoS?

Both ways are desireable. AoS can provide portability to other APIs and easier to setup, while SoA allows to update individual inputs in a cache-efficient manner, when others remain static.

6) How do we make developers aware of the memory requirements of implementation-dependent data used for the generated commands?

Make the API explicit and introduce a preprocess VkBuffer. Developers have to allocate it using vkGetGeneratedCommandsMemoryRequirementsNV.

In the NVX version the requirements were hidden implicitly as part of the command buffer reservation process, however as the memory requirements can be substantial, we want to give developers the ability to budget the memory themselves. By lowering the maxSequencesCount the memory consumption can be reduced. Furthermore reuse of the memory is possible, for example for doing explicit preprocessing and execution in a ping-pong fashion.

The actual buffer size is implementation dependent and may be zero, i.e. not always required.

When making use of Graphics Shader Groups, the programs should behave similar with regards to vertex inputs, clipping and culling outputs of the geometry stage, as well as sample shading behavior in fragment shaders, to reduce the amount of the worst-case memory approximation.

7) Should we allow additional per-sequence dynamic state changes?

Yes

Introduced a lightweight indirect state flag VkIndirectStateFlagBitsNV. So far only switching front face winding state is exposed. Especially in CAD/DCC mirrored transforms that require such changes are common, and similar flexibility is given in the ray tracing instance description.

The flag could be extended further, for example to switch between primitive-lists or -strips, or
make other state modifications.

Furthermore, as new tokens can be added easily, future extension could add the ability to change any `VkDynamicState`.

8) How do we allow re-using already “generated” `indirectCommands`?

Expose a `preprocessBuffer` to reuse implementation-dependencyFlags data. Set the `isPreprocessed` to true in `vkCmdExecuteGeneratedCommandsNV`.

9) Under which conditions is `vkCmdExecuteGeneratedCommandsNV` legal?

It behaves like a regular draw call command.

10) Is `vkCmdPreprocessGeneratedCommandsNV` copying the input data or referencing it?

There are multiple implementations possible:

- one could have some emulation code that parses the inputs, and generates an output command buffer, therefore copying the inputs.
- one could just reference the inputs, and have the processing done in pipe at execution time.

If the data is mandated to be copied, then it puts a penalty on implementation that could process the inputs directly in pipe. If the data is “referenced”, then it allows both types of implementation.

The inputs are “referenced”, and must not be modified after the call to `vkCmdExecuteGeneratedCommandsNV` has completed.

11) Which buffer usage flags are required for the buffers referenced by `VkGeneratedCommandsInfoNV`?

Reuse existing `VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT`

- `VkGeneratedCommandsInfoNV::preprocessBuffer`
- `VkGeneratedCommandsInfoNV::sequencesCountBuffer`
- `VkGeneratedCommandsInfoNV::sequencesIndexBuffer`
- `VkIndirectCommandsStreamNV::buffer`

12) In which pipeline stage does the device generated command expansion happen?

`vkCmdPreprocessGeneratedCommandsNV` is treated as if it occurs in a separate logical pipeline from either graphics or compute, and that pipeline only includes `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT`, a new stage `VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV`, and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`. This new stage has two corresponding new access types, `VK_ACCESS_COMMAND_PREPROCESS_READ_BIT_NV` and `VK_ACCESS_COMMAND_PREPROCESS_WRITE_BIT_NV`, used to synchronize reading the buffer inputs and writing the preprocess memory output.

The generated output written in the preprocess buffer memory by `vkCmdExecuteGeneratedCommandsNV` is considered to be consumed by the `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT` pipeline stage.
Thus, to synchronize from writing the input buffers to preprocessing via \texttt{vkCmdPreprocessGeneratedCommandsNV}, use:

- \texttt{dstStageMask} = \texttt{VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV}
- \texttt{dstAccessMask} = \texttt{VK_ACCESS_COMMAND_PREPROCESS_READ_BIT_NV}

To synchronize from \texttt{vkCmdPreprocessGeneratedCommandsNV} to executing the generated commands by \texttt{vkCmdExecuteGeneratedCommandsNV}, use:

- \texttt{srcStageMask} = \texttt{VK_PIPELINE_STAGE_COMMAND_PREPROCESS_BIT_NV}
- \texttt{srcAccessMask} = \texttt{VK_ACCESS_COMMAND_PREPROCESS_WRITE_BIT_NV}
- \texttt{dstStageMask} = \texttt{VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT}
- \texttt{dstAccessMask} = \texttt{VK_ACCESS_INDIRECT_COMMAND_READ_BIT}

When \texttt{vkCmdExecuteGeneratedCommandsNV} is used with a \texttt{isPreprocessed} of \texttt{VK_FALSE}, the generated commands are implicitly preprocessed, therefore one only needs to synchronize the inputs via:

- \texttt{dstStageMask} = \texttt{VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT}
- \texttt{dstAccessMask} = \texttt{VK_ACCESS_INDIRECT_COMMAND_READ_BIT}

13) What if most token data is “static”, but we frequently want to render a subsection?

Added “sequencesIndexBuffer”. This allows to easier sort and filter what should actually be executed.

14) What are the changes compared to the previous NVX extension?

- Compute dispatch support was removed (was never implemented in drivers). There are different approaches how dispatching from the device should work, hence we defer this to a future extension.
- The \texttt{ObjectTableNVX} was replaced by using physical buffer addresses and introducing Shader Groups for the graphics pipeline.
- Less state changes are possible overall, but the important operations are still there (reduces complexity of implementation).
- The API was redesigned so all inputs must be passed at both preprocessing and execution time (this was implicit in NVX, now it is explicit)
- The reservation of intermediate command space is now mandatory and explicit through a preprocess buffer.
- The \texttt{VkIndirectStateFlagBitsNV} were introduced

15) When porting from other APIs, their indirect buffers may use different enums, for example for index buffer types. How to solve this?

Added “pIndexTypeValues” to map custom \texttt{uint32_t} values to corresponding \texttt{VkIndexType}.

16) Do we need more shader group state overrides?
The NVX version allowed all PSO states to be different, however as the goal is not to replace all state setup, but focus on highly-frequent state changes for drawing lots of objects, we reduced the amount of state overrides. Especially VkPipelineLayout as well as VkRenderPass configuration should be left static, the rest is still open for discussion.

The current focus is just to allow VertexInput changes as well as shaders, while all shader groups use the same shader stages.

Too much flexibility will increase the test coverage requirement as well. However, further extensions could allow more dynamic state as well.

17) Do we need more detailed physical device feature queries/enables?

An EXT version would need detailed implementor feedback to come up with a good set of features. Please contact us if you are interested, we are happy to make more features optional, or add further restrictions to reduce the minimum feature set of an EXT.

18) Is there an interaction with VK_KHR_pipeline_library planned?

Yes, a future version of this extension will detail the interaction, once VK_KHR_pipeline_library is no longer provisional.

Example Code

Open-Source samples illustrating the usage of the extension can be found at the following location (may not yet exist at time of writing):

https://github.com/nvpro-samples/vk_device_generated_cmds

Version History

- Revision 1, 2020-02-20 (Christoph Kubisch)
  - Initial version
- Revision 2, 2020-03-09 (Christoph Kubisch)
  - Remove VK_EXT_debug_report interactions
- Revision 3, 2020-03-09 (Christoph Kubisch)
  - Fix naming VkPhysicalDeviceGenerated to VkPhysicalDeviceDeviceGenerated

VK_NV_fill_rectangle

Name String

VK_NV_fill_rectangle

Extension Type

Device extension

Registered Extension Number

154
**Revision**
1

**Extension and Version Dependencies**
- Requires Vulkan 1.0

**Contact**
- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

**Other Extension Metadata**

**Last Modified Date**
2017-05-22

**Contributors**
- Jeff Bolz, NVIDIA

**Description**
This extension adds a new `VkPolygonMode` enum where a triangle is rasterized by computing and filling its axis-aligned screen-space bounding box, disregarding the actual triangle edges. This can be useful for drawing a rectangle without being split into two triangles with an internal edge. It is also useful to minimize the number of primitives that need to be drawn, particularly for a user interface.

**New Enum Constants**
- `VK_NV_FILL_RECTANGLE_EXTENSION_NAME`
- `VK_NV_FILL_RECTANGLE_SPEC_VERSION`
- Extending `VkPolygonMode`:
  - `VK_POLYGON_MODE_FILL_RECTANGLE_NV`

**Version History**
- Revision 1, 2017-05-22 (Jeff Bolz)
  - Internal revisions

**VK_NV_fragment_coverage_to_color**

**Name String**
`VK_NV_fragment_coverage_to_color`

**Extension Type**
Device extension
Description

This extension allows the fragment coverage value, represented as an integer bitmask, to be substituted for a color output being written to a single-component color attachment with integer components (e.g. `VK_FORMAT_R8_UINT`). The functionality provided by this extension is different from simply writing the `SampleMask` fragment shader output, in that the coverage value written to the framebuffer is taken after stencil test and depth test, as well as after fragment operations such as alpha-to-coverage.

This functionality may be useful for deferred rendering algorithms, where the second pass needs to know which samples belong to which original fragments.

New Structures

- Extending `VkPipelineMultisampleStateCreateInfo`:
  - `VkPipelineCoverageToColorStateCreateInfoNV`

New Bitmasks

- `VkPipelineCoverageToColorStateCreateFlagsNV`

New Enum Constants

- `VK_NV_FRAGMENT_COVERAGE_TO_COLOR_EXTENSION_NAME`
- `VK_NV_FRAGMENT_COVERAGE_TO_COLOR_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_TO_COLOR_STATE_CREATE_INFO_NV`
VK_NV_fragment_shader_barycentric

Name String
VK_NV_fragment_shader_barycentric

Extension Type
Device extension

Registered Extension Number
204

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact
• Pat Brown @nvpbrown

Other Extension Metadata

Last Modified Date
2018-08-03

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_NV_fragment_shader_barycentric
• This extension provides API support for GL_NV_fragment_shader_barycentric

Contributors
• Pat Brown, NVIDIA
• Daniel Koch, NVIDIA

Description
This extension adds support for the following SPIR-V extension in Vulkan:
• SPV_NV_fragment_shader_barycentric
The extension provides access to three additional fragment shader variable decorations in SPIR-V:

- **PerVertexNV**, which indicates that a fragment shader input will not have interpolated values, but instead must be accessed with an extra array index that identifies one of the vertices of the primitive producing the fragment

- **BaryCoordNV**, which indicates that the variable is a three-component floating-point vector holding barycentric weights for the fragment produced using perspective interpolation

- **BaryCoordNoPerspNV**, which indicates that the variable is a three-component floating-point vector holding barycentric weights for the fragment produced using linear interpolation

When using GLSL source-based shader languages, the following variables from **GL_NV_fragment_shader_barycentric** maps to these SPIR-V built-in decorations:

- `in vec3 gl_BaryCoordNV;` → **BaryCoordNV**
- `in vec3 gl_BaryCoordNoPerspNV;` → **BaryCoordNoPerspNV**

GLSL variables declared using the **__pervertexNV** GLSL qualifier are expected to be decorated with **PerVertexNV** in SPIR-V.

**New Structures**

- Extending **VkPhysicalDeviceFeatures2, VkDeviceCreateInfo**:
  - **VkPhysicalDeviceFragmentShaderBarycentricFeaturesNV**

**New Enum Constants**

- **VK_NV_FRAGMENT_SHADER_BARYCENTRIC_EXTENSION_NAME**
- **VK_NV_FRAGMENT_SHADER_BARYCENTRIC_SPEC_VERSION**

- Extending **VkStructureType**:
  - **VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADER_BARYCENTRIC_FEATURES_NV**

**New Built-In Variables**

- **BaryCoordNV**
- **BaryCoordNoPerspNV**

**New SPIR-V Decorations**

- **PerVertexNV**

**New SPIR-V Capabilities**

- **FragmentBarycentricNV**
Issues

(1) The AMD_shader_explicit_vertex_parameter extension provides similar functionality. Why write a new extension, and how is this extension different?

**RESOLVED:** For the purposes of Vulkan/SPIR-V, we chose to implement a separate extension due to several functional differences.

First, the hardware supporting this extension can provide a three-component barycentric weight vector for variables decorated with `BaryCoordNV`, while variables decorated with `BaryCoordSmoothAMD` provide only two components. In some cases, it may be more efficient to explicitly interpolate an attribute via:

```c
float value = (baryCoordNV.x * v[0].attrib +
               baryCoordNV.y * v[1].attrib +
               baryCoordNV.z * v[2].attrib);
```

instead of

```c
float value = (baryCoordSmoothAMD.x * (v[0].attrib - v[2].attrib) +
               baryCoordSmoothAMD.y * (v[1].attrib - v[2].attrib) +
               v[2].attrib);
```

Additionally, the semantics of the decoration `BaryCoordPullModelAMD` do not appear to map to anything supported by the initial hardware implementation of this extension.

This extension provides a smaller number of decorations than the AMD extension, as we expect that shaders could derive variables decorated with things like `BaryCoordNoPerspCentroidAMD` with explicit attribute interpolation instructions. One other relevant difference is that explicit per-vertex attribute access using this extension does not require a constant vertex number.

(2) Why do the built-in SPIR-V decorations for this extension include two separate built-ins `BaryCoordNV` and `BaryCoordNoPerspNV` when a “no perspective” variable could be decorated with `BaryCoordNV` and `NoPerspective`?

**RESOLVED:** The SPIR-V extension for this feature chose to mirror the behavior of the GLSL extension, which provides two built-in variables. Additionally, it’s not clear that its a good idea (or even legal) to have two variables using the “same attribute”, but with different interpolation modifiers.

Version History

- Revision 1, 2018-08-03 (Pat Brown)
  - Internal revisions
VK_NV_fragment_shading_rate Enums

Name String
  VK_NV_fragment_shading_rate Enums

Extension Type
  Device extension

Registered Extension Number
  327

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_fragment_shading_rate

Contact
  • Pat Brown @nvpbrown

Other Extension Metadata

Last Modified Date
  2020-09-02

Contributors
  • Pat Brown, NVIDIA
  • Jeff Bolz, NVIDIA

Description
This extension builds on the fragment shading rate functionality provided by the VK_KHR_fragment_shading_rate extension, adding support for “supersample” fragment shading rates that trigger multiple fragment shader invocations per pixel as well as a “no invocations” shading rate that discards any portions of a primitive that would use that shading rate.

New Commands
  • vkCmdSetFragmentShadingRateEnumNV

New Structures
  • Extending VkGraphicsPipelineCreateInfo:
    ◦ VkPipelineFragmentShadingRateEnumStateCreateInfoNV
  • Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
New Enums

- VkFragmentShadingRateNV
- VkFragmentShadingRateTypeNV

New Enum Constants

- VK_NV_FRAGMENT_SHADING_RATE_ENUMS_EXTENSION_NAME
- VK_NV_FRAGMENT_SHADING_RATE_ENUMS_SPEC_VERSION

Extending VkStructureType:

- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_ENUMS_FEATURES_NV
- VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FRAGMENT_SHADING_RATE_ENUMS_PROPERTIES_NV
- VK_STRUCTURE_TYPE_PIPELINE_FRAGMENT_SHADING_RATE_ENUM_STATE_CREATE_INFO_NV

Issues

1. Why was this extension created? How should it be named?

   **RESOLVED:** The primary goal of this extension was to expose support for supersample and “no invocations” shading rates, which are supported by the VK_NV_shading_rate_image extension but not by VK_KHR_fragment_shading_rate. Because VK_KHR_fragment_shading_rate specifies the primitive shading rate using a fragment size in pixels, it lacks a good way to specify supersample rates. To deal with this, we defined enums covering shading rates supported by the KHR extension as well as the new shading rates and added structures and APIs accepting shading rate enums instead of fragment sizes.

   Since this extension adds two different types of shading rates, both expressed using enums, we chose the extension name VK_NV_fragment_shading_rateEnums.

2. Is this a standalone extension?

   **RESOLVED:** No, this extension requires VK_KHR_fragment_shading_rate. In order to use the features of this extension, applications must enable the relevant features of KHR extension.

3. How are the shading rate enums used, and how were the enum values assigned?

   **RESOLVED:** The shading rates supported by the enums in this extension are accepted as pipeline, primitive, and attachment shading rates and behave identically. For the shading rates also supported by the KHR extension, the values assigned to the corresponding enums are identical to the values already used for the primitive and attachment shading rates in the KHR extension. For those enums, bits 0 and 1 specify the base two logarithm of the fragment height and bits 2 and 3 specify the base two logarithm of the fragment width. For the new shading
rates added by this extension, we chose to use 11 through 14 (10 plus the base two logarithm of
the invocation count) for the supersample rates and 15 for the “no invocations” rate. None of
those values are supported as primitive or attachment shading rates by the KHR extension.

4. Between this extension, VK_KHR_fragment_shading_rate, and VK_NV_shading_rate_image, there
are three different ways to specify shading rate state in a pipeline. How should we handle this?

**RESOLVED:** We don’t allow the concurrent use of VK_NV_shading_rate_image and
VK_KHR_fragment_shading_rate; it is an error to enable shading rate features from both
extensions. But we do allow applications to enable this extension together with
VK_KHR_fragment_shading_rate together. While we expect that applications will never attach
pipeline CreateInfo structures for both this extension and the KHR extension concurrently,
Vulkan doesn’t have any precedent forbidding such behavior and instead typically treats a
pipeline created without an extension-specific CreateInfo structure as equivalent to one
containing default values specified by the extension. Rather than adding such a rule considering
the presence or absence of our new CreateInfo structure, we instead included a `shadingRateType`
member to `VkPipelineFragmentShadingRateEnumStateCreateInfoNV` that selects between using
state specified by that structure and state specified by `VkPipelineFragmentShadingRateStateCreateInfoKHR`.

**Version History**

- Revision 1, 2020-09-02 (pbrown)
  - Internal revisions

**VK_NV_framebuffer_mixed_samples**

**Name String**

VK_NV_framebuffer_mixed_samples

**Extension Type**

Device extension

**Registered Extension Number**

153

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

**Other Extension Metadata**
Description

This extension allows multisample rendering with a raster and depth/stencil sample count that is larger than the color sample count. Rasterization and the results of the depth and stencil tests together determine the portion of a pixel that is “covered”. It can be useful to evaluate coverage at a higher frequency than color samples are stored. This coverage is then “reduced” to a collection of covered color samples, each having an opacity value corresponding to the fraction of the color sample covered. The opacity can optionally be blended into individual color samples.

Rendering with fewer color samples than depth/stencil samples greatly reduces the amount of memory and bandwidth consumed by the color buffer. However, converting the coverage values into opacity introduces artifacts where triangles share edges and may not be suitable for normal triangle mesh rendering.

One expected use case for this functionality is Stencil-then-Cover path rendering (similar to the OpenGL GL_NV_path_rendering extension). The stencil step determines the coverage (in the stencil buffer) for an entire path at the higher sample frequency, and then the cover step draws the path into the lower frequency color buffer using the coverage information to antialias path edges. With this two-step process, internal edges are fully covered when antialiasing is applied and there is no corruption on these edges.

The key features of this extension are:

- It allows render pass and framebuffer objects to be created where the number of samples in the depth/stencil attachment in a subpass is a multiple of the number of samples in the color attachments in the subpass.

- A coverage reduction step is added to Fragment Operations which converts a set of covered raster/depth/stencil samples to a set of color samples that perform blending and color writes. The coverage reduction step also includes an optional coverage modulation step, multiplying color values by a fractional opacity corresponding to the number of associated raster/depth/stencil samples covered.

New Structures

- Extending VkPipelineMultisampleStateCreateInfo:
  - VkPipelineCoverageModulationStateCreateInfoNV

New Enums

- VkCoverageModulationModeNV
New Bitmasks

- VkPipelineCoverageModulationStateCreateFlagsNV

New Enum Constants

- VK_NV_FRAMEBUFFER_MIXED_SAMPLES_EXTENSION_NAME
- VK_NV_FRAMEBUFFER_MIXED_SAMPLES_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PIPELINE_COVERAGE_MODULATION_STATE_CREATE_INFO_NV

Version History

- Revision 1, 2017-06-04 (Jeff Bolz)
  - Internal revisions

VK_NV_geometry_shader_passthrough

Name String

VK_NV_geometry_shader_passthrough

Extension Type

Device extension

Registered Extension Number

96

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Daniel Koch dgkoch

Other Extension Metadata

Last Modified Date

2017-02-15

Interactions and External Dependencies

- This extension requires SPV_NV_geometry_shader_passthrough
- This extension provides API support for GL_NV_geometry_shader_passthrough
- This extension requires the geometryShader feature.
Contributors

- Piers Daniell, NVIDIA
- Jeff Bolz, NVIDIA

Description

This extension adds support for the following SPIR-V extension in Vulkan:

- **SPV_NV_geometry_shader_passthrough**

Geometry shaders provide the ability for applications to process each primitive sent through the graphics pipeline using a programmable shader. However, one common use case treats them largely as a “passthrough”. In this use case, the bulk of the geometry shader code simply copies inputs from each vertex of the input primitive to corresponding outputs in the vertices of the output primitive. Such shaders might also compute values for additional built-in or user-defined per-primitive attributes (e.g., *Layer*) to be assigned to all the vertices of the output primitive.

This extension provides access to the *PassthroughNV* decoration under the *GeometryShaderPassthroughNV* capability. Adding this to a geometry shader input variable specifies that the values of this input are copied to the corresponding vertex of the output primitive.

When using GLSL source-based shading languages, the *passthrough* layout qualifier from *GL_NV_geometry_shader_passthrough* maps to the *PassthroughNV* decoration. To use the *passthrough* layout, in GLSL the *GL_NV_geometry_shader_passthrough* extension must be enabled. Behaviour is described in the *GL_NV_geometry_shader_passthrough* extension specification.

New Enum Constants

- **VK_NV_GEOMETRY_SHADER_PASSTHROUGH_EXTENSION_NAME**
- **VK_NV_GEOMETRY_SHADER_PASSTHROUGH_SPEC_VERSION**

New Variable Decoration

- **PassthroughNV** in Geometry Shader Passthrough

New SPIR-V Capabilities

- **GeometryShaderPassthroughNV**

Issues

1) Should we require or allow a passthrough geometry shader to specify the output layout qualifiers for the output primitive type and maximum vertex count in the SPIR-V?

**RESOLVED:** Yes they should be required in the SPIR-V. Per *GL_NV_geometry_shader_passthrough* they are not permitted in the GLSL source shader, but SPIR-V is lower-level. It is straightforward for the GLSL compiler to infer them from the input primitive type and to explicitly emit them in the SPIR-V according to the following table.
2) How does interface matching work with passsthrough geometry shaders?

**RESOLVED:** This is described in [Passthrough Interface Matching](#). In GL when using passthough geometry shaders in separable mode, all inputs must also be explicitly assigned location layout qualifiers. In Vulkan all SPIR-V shader inputs (except built-ins) must also have location decorations specified. Redeclarations of built-in variables that add the passthrough layout qualifier are exempted from the rule requiring location assignment because built-in variables do not have locations and are matched by `BuiltIn` decoration.

### Sample Code

Consider the following simple geometry shader in unextended GLSL:

```glsl
layout(triangles) in;
layout(triangle_strip) out;
layout(max_vertices=3) out;

in Inputs {
  vec2 texcoord;
  vec4 baseColor;
} v_in[];
out Outputs {
  vec2 texcoord;
  vec4 baseColor;
};

void main()
{
  int layer = compute_layer();
  for (int i = 0; i < 3; i++) {
    gl_Position = gl_in[i].gl_Position;
    texcoord = v_in[i].texcoord;
    baseColor = v_in[i].baseColor;
    gl_Layer = layer;
    EmitVertex();
  }
}
```

In this shader, the inputs `gl_Position`, `Inputs.texcoord`, and `Inputs.baseColor` are simply copied from the input vertex to the corresponding output vertex. The only “interesting” work done by the geometry shader is computing and emitting a `gl_Layer` value for the primitive.

The following geometry shader, using this extension, is equivalent:
#extension GL_NV_geometry_shader_passthrough : require

layout(triangles) in;
// No output primitive layout qualifiers required.

// Redeclare gl_PerVertex to pass through "gl_Position".
layout(passthrough) in gl_PerVertex {
  vec4 gl_Position;
} gl_in[];

// Declare "Inputs" with "passthrough" to automatically copy members.
layout(passthrough) in Inputs {
  vec2 texcoord;
  vec4 baseColor;
} v_in[];

// No output block declaration required.

void main()
{
    // The shader simply computes and writes gl_Layer. We don't
    // loop over three vertices or call EmitVertex().
    gl_Layer = compute_layer();
}

Version History

• Revision 1, 2017-02-15 (Daniel Koch)
  ◦ Internal revisions

VK_NV_inherited_viewport_scissor

Name String
    VK_NV_inherited_viewport_scissor

Extension Type
    Device extension

Registered Extension Number
    279

Revision
    1

Extension and Version Dependencies
    • Requires Vulkan 1.0
Description

This extension adds the ability for a secondary command buffer to inherit the dynamic viewport and scissor state from a primary command buffer, or a previous secondary command buffer executed within the same `vkCmdExecuteCommands` call. It addresses a frequent scenario in applications that deal with window resizing and want to improve utilization of re-usable secondary command buffers. The functionality is provided through `VkCommandBufferInheritanceViewportScissorInfoNV`. Viewport inheritance is effectively limited to the 2D rectangle; secondary command buffers must re-specify the inherited depth range values.

New Structures

- Extending `VkCommandBufferInheritanceInfo`:
  - `VkCommandBufferInheritanceViewportScissorInfoNV`
- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceInheritedViewportScissorFeaturesNV`

New Enum Constants

- `VK_NV_INHERITED_VIEWPORT_SCISSOR_EXTENSION_NAME`
- `VK_NV_INHERITED_VIEWPORT_SCISSOR_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_VIEWPORT_SCISSOR_INFO_NV`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_INHERITED_VIEWPORT_SCISSOR_FEATURES_NV`

Issues

(1) Why are viewport depth values configured in the `VkCommandBufferInheritanceViewportScissorInfoNV` struct, rather than by a `vkCmd...` function?
DISCUSSION:

We considered both adding a new `vkCmdSetViewportDepthNV` function, and modifying `vkCmdSetViewport` to ignore the \( x, y, \text{width}, \text{and height} \) values when called with a secondary command buffer that activates this extension. 

The primary design considerations for this extension are debuggability and easy integration into existing applications. The main issue with adding a new `vkCmdSetViewportDepthNV` function is reducing ease-of-integration. A new function pointer will have to be loaded, but more importantly, a new function would require changes to be supported in graphics debuggers; this would delay widespread adoption of the extension.

The proposal to modify `vkCmdSetViewport` would avoid these issues. However, we expect that the intent of applications using this extension is to have the viewport values used for drawing exactly match the inherited values; thus, it would be better for debuggability if no function for modifying the viewport depth alone is provided. By specifying viewport depth values when starting secondary command buffer recording, and requiring the specified depth values to match the inherited depth values, we allow for validation layers that flag depth changes as errors.

This design also better matches the hardware model. In fact, there is no need to re-execute a depth-setting command. The graphics device retains the viewport depth state; it’s the CPU-side state of `VkCommandBuffer` that must be re-initialized.

(2) Why are viewport depth values specified as a partial `VkViewport` struct, rather than a leaner depth-only struct?

DISCUSSION:

We considered adding a new `VkViewportDepthNV` struct containing only `minDepth` and `maxDepth`. However, as application developers would need to maintain both a `VK_NV_inherited_viewport_scissor` code path and a fallback code path (at least in the short term), we ultimately chose to continue using the existing `VkViewport` structure. Doing so would allow application developers to reuse the same `VkViewport` array for both code paths, rather than constructing separate `VkViewportDepthNV` and `VkViewport` arrays for each code path.

Version History

- Revision 1, 2020-02-04 (David Zhao Akeley)
  - Internal revisions

VK_NV_mesh_shader

Name String

`VK_NV_mesh_shader`

Extension Type

Device extension
Registered Extension Number
203

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact
• Christoph Kubisch pixeljetstream

Other Extension Metadata

Last Modified Date
2018-07-19

Interactions and External Dependencies
• This extension requires SPV_NV_mesh_shader
• This extension provides API support for GLSL_NV_mesh_shader

Contributors
• Pat Brown, NVIDIA
• Jeff Bolz, NVIDIA
• Daniel Koch, NVIDIA
• Piers Daniell, NVIDIA
• Pierre Boudier, NVIDIA

Description
This extension provides a new mechanism allowing applications to generate collections of geometric primitives via programmable mesh shading. It is an alternative to the existing programmable primitive shading pipeline, which relied on generating input primitives by a fixed function assembler as well as fixed function vertex fetch.

There are new programmable shader types—the task and mesh shader—to generate these collections to be processed by fixed-function primitive assembly and rasterization logic. When the task and mesh shaders are dispatched, they replace the standard programmable vertex processing pipeline, including vertex array attribute fetching, vertex shader processing, tessellation, and the geometry shader processing.

This extension also adds support for the following SPIR-V extension in Vulkan:

• SPV_NV_mesh_shader
New Commands

- vkCmdDrawMeshTasksIndirectCountNV
- vkCmdDrawMeshTasksIndirectNV
- vkCmdDrawMeshTasksNV

New Structures

- VkDrawMeshTasksIndirectCommandNV
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceMeshShaderFeaturesNV
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceMeshShaderPropertiesNV

New Enum Constants

- VK_NV_MESH_SHADER_EXTENSION_NAME
- VK_NV_MESH_SHADER_SPEC_VERSION
- Extending VkPipelineStageFlagBits:
  - VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV
  - VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV
- Extending VkShaderStageFlagBits:
  - VK_SHADER_STAGE_MESH_BIT_NV
  - VK_SHADER_STAGE_TASK_BIT_NV
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_FEATURES_NV
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MESH_SHADER_PROPERTIES_NV

New or Modified Built-In Variables

- TaskCountNV
- PrimitiveCountNV
- PrimitiveIndicesNV
- ClipDistancePerViewNV
- CullDistancePerViewNV
- LayerPerViewNV
- MeshViewCountNV
- MeshViewIndicesNV
- (modified) Position
• (modified)PointSize
• (modified)ClipDistance
• (modified)CullDistance
• (modified)PrimitiveId
• (modified)Layer
• (modified)ViewportIndex
• (modified)WorkgroupSize
• (modified)WorkgroupId
• (modified)LocalInvocationId
• (modified)GlobalInvocationId
• (modified)LocalInvocationIndex
• (modified)DrawIndex
• (modified)ViewportMaskNV
• (modified)PositionPerViewNV
• (modified)ViewportMaskPerViewNV

New SPIR-V Capability

• MeshShadingNV

Issues

1. How to name this extension?

RESOLVED: VK_NV_mesh_shader

Other options considered:

◦ VK_NV_mesh_shading
◦ VK_NV_programmable_mesh_shading
◦ VK_NV_primitive_group_shading
◦ VK_NV_grouped_drawing

2. Do we need a new VkPrimitiveTopology?

RESOLVED: No. We skip the InputAssembler stage.

3. Should we allow Instancing?

RESOLVED: No. There is no fixed function input, other than the IDs. However, allow offsetting with a “first” value.

4. Should we use existing vkCmdDraw or introduce new functions?
RESOLVED: Introduce new functions.

New functions make it easier to separate from “programmable primitive shading” chapter, less “dual use” language about existing functions having alternative behavior. The text around the existing “draws” is heavily based around emitting vertices.

5. If new functions, how to name?

RESOLVED: CmdDrawMeshTasks*

Other options considered:

◦ CmdDrawMeshed
◦ CmdDrawTasked
◦ CmdDrawGrouped

6. Should VK_SHADER_STAGE_ALL_GRAPHICS be updated to include the new stages?

RESOLVED: No. If an application were to be recompiled with headers that include additional shader stage bits in VK_SHADER_STAGE_ALL_GRAPHICS, then the previously valid application would no longer be valid on implementations that don’t support mesh or task shaders. This means the change would not be backwards compatible. It’s too bad VkShaderStageFlagBits doesn’t have a dedicated “all supported graphics stages” bit like VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT, which would have avoided this problem.

Version History

• Revision 1, 2018-07-19 (Christoph Kubisch, Daniel Koch)
  ◦ Internal revisions

VK_NV_ray_tracing

Name String

VK_NV_ray_tracing

Extension Type

Device extension

Registered Extension Number

166

Revision

3

Extension and Version Dependencies

• Requires Vulkan 1.0

• Requires VK_KHR_get_physical_device_properties2
• Requires VK_KHR_get_memory_requirements2

Contact
• Eric Werness ewerness

Other Extension Metadata

Last Modified Date
2018-11-20

Interactions and External Dependencies
• This extension requires SPV_NV_ray_tracing
• This extension provides API support for GL_NV_ray_tracing

Contributors
• Eric Werness, NVIDIA
• Ashwin Lele, NVIDIA
• Robert Stepinski, NVIDIA
• Nuno Subtil, NVIDIA
• Christoph Kubisch, NVIDIA
• Martin Stich, NVIDIA
• Daniel Koch, NVIDIA
• Jeff Bolz, NVIDIA
• Joshua Barczak, Intel
• Tobias Hector, AMD
• Henrik Rydgard, NVIDIA
• Pascal Gautron, NVIDIA

Description
Rasterization has been the dominant method to produce interactive graphics, but increasing performance of graphics hardware has made ray tracing a viable option for interactive rendering. Being able to integrate ray tracing with traditional rasterization makes it easier for applications to incrementally add ray traced effects to existing applications or to do hybrid approaches with rasterization for primary visibility and ray tracing for secondary queries.

To enable ray tracing, this extension adds a few different categories of new functionality:

• Acceleration structure objects and build commands
• A new pipeline type with new shader domains
• An indirection table to link shader groups with acceleration structure items

This extension adds support for the following SPIR-V extension in Vulkan:
SPV_NV_ray_tracing

New Object Types

- VkAccelerationStructureNV

New Commands

- vkBindAccelerationStructureMemoryNV
- vkCmdBuildAccelerationStructureNV
- vkCmdCopyAccelerationStructureNV
- vkCmdTraceRaysNV
- vkCmdWriteAccelerationStructuresPropertiesNV
- vkCompileDeferredNV
- vkCreateAccelerationStructureNV
- vkCreateRayTracingPipelinesNV
- vkDestroyAccelerationStructureNV
- vkGetAccelerationStructureHandleNV
- vkGetAccelerationStructureMemoryRequirementsNV
- vkGetRayTracingShaderGroupHandlesNV

New Structures

- VkAabbPositionsNV
- VkAccelerationStructureCreateInfoNV
- VkAccelerationStructureInfoNV
- VkAccelerationStructureInstanceNV
- VkAccelerationStructureMemoryRequirementsInfoNV
- VkBindAccelerationStructureMemoryInfoNV
- VkGeometryAABBNV
- VkGeometryDataNV
- VkGeometryNV
- VkGeometryTrianglesNV
- VkMemoryRequirements2KHR
- VkRayTracingPipelineCreateInfoNV
- VkRayTracingShaderGroupCreateInfoNV
- VkTransformMatrixNV

Extending VkPhysicalDeviceProperties2:
• VkPhysicalDeviceRayTracingPropertiesNV

Extending VkWriteDescriptorSet:
• VkWriteDescriptorSetAccelerationStructureNV

New Enums
• VkAccelerationStructureMemoryRequirementsTypeNV
• VkAccelerationStructureTypeNV
• VkBuildAccelerationStructureFlagBitsNV
• VkCopyAccelerationStructureModeNV
• VkGeometryFlagBitsNV
• VkGeometryInstanceFlagBitsNV
• VkGeometryTypeNV
• VkRayTracingShaderGroupTypeNV

New Bitmasks
• VkBuildAccelerationStructureFlagsNV
• VkGeometryFlagsNV
• VkGeometryInstanceFlagsNV

New Enum Constants
• VK_NV_RAY_TRACING_EXTENSION_NAME
• VK_NV_RAY_TRACING_SPEC_VERSION
• VK_SHADER_UNUSED_NV

Extending VkAccelerationStructureTypeKHR:
• VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_NV
• VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_NV

Extending VkAccessFlagBits:
• VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV
• VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV

Extending VkBufferUsageFlagBits:
• VK_BUFFER_USAGE_RAY_TRACING_BIT_NV

Extending VkBuildAccelerationStructureFlagBitsKHR:
• VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV
• VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV
• VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_NV
- \texttt{VK\_BUILD\_ACCELERATION\_STRUCTURE\_PREFER\_FAST\_BUILD\_BIT\_NV}
- \texttt{VK\_BUILD\_ACCELERATION\_STRUCTURE\_PREFER\_FAST\_TRACE\_BIT\_NV}

- **Extending \texttt{VkCopyAccelerationStructureModeKHR}:**
  - \texttt{VK\_COPY\_ACCELERATION\_STRUCTURE\_MODE\_CLONE\_NV}
  - \texttt{VK\_COPY\_ACCELERATION\_STRUCTURE\_MODE\_COMPACT\_NV}

- **Extending \texttt{VkDebugReportObjectTypeEXT}:**
  - \texttt{VK\_DEBUG\_REPORT\_OBJECT\_TYPE\_ACCELERATION\_STRUCTURE\_NV\_EXT}

- **Extending \texttt{VkDescriptorType}:**
  - \texttt{VK\_DESCRIPTOR\_TYPE\_ACCELERATION\_STRUCTURE\_NV}

- **Extending \texttt{VkGeometryFlagBitsKHR}:**
  - \texttt{VK\_GEOMETRY\_NO\_DUPLICATE\_ANY\_HIT\_INVOCATION\_BIT\_NV}
  - \texttt{VK\_GEOMETRY\_OPAQUE\_BIT\_NV}

- **Extending \texttt{VkGeometryInstanceFlagBitsKHR}:**
  - \texttt{VK\_GEOMETRY\_INSTANCE\_FORCE\_NO\_OPAQUE\_BIT\_NV}
  - \texttt{VK\_GEOMETRY\_INSTANCE\_FORCE\_OPAQUE\_BIT\_NV}
  - \texttt{VK\_GEOMETRY\_INSTANCE\_TRIANGLE\_CULL\_DISABLE\_BIT\_NV}
  - \texttt{VK\_GEOMETRY\_INSTANCE\_TRIANGLE\_FRONT\_COUNTERCLOCKWISE\_BIT\_NV}

- **Extending \texttt{VkGeometryTypeKHR}:**
  - \texttt{VK\_GEOMETRY\_TYPE\_AABBS\_NV}
  - \texttt{VK\_GEOMETRY\_TYPE\_TRIANGLES\_NV}

- **Extending \texttt{VkIndexType}:**
  - \texttt{VK\_INDEX\_TYPE\_NONE\_NV}

- **Extending \texttt{VkObjectType}:**
  - \texttt{VK\_OBJECT\_TYPE\_ACCELERATION\_STRUCTURE\_NV}

- **Extending \texttt{VkPipelineBindPoint}:**
  - \texttt{VK\_PIPELINE\_BIND\_POINT\_RAY\_TRACING\_NV}

- **Extending \texttt{VkPipelineCreateFlagBits}:**
  - \texttt{VK\_PIPELINE\_CREATE\_DEFER\_COMPILE\_BIT\_NV}

- **Extending \texttt{VkPipelineStageFlagBits}:**
  - \texttt{VK\_PIPELINE\_STAGE\_ACCELERATION\_STRUCTURE\_BUILD\_BIT\_NV}
  - \texttt{VK\_PIPELINE\_STAGE\_RAY\_TRACING\_SHADER\_BIT\_NV}

- **Extending \texttt{VkQueryType}:**
  - \texttt{VK\_QUERY\_TYPE\_ACCELERATION\_STRUCTURE\_COMPACTED\_SIZE\_NV}

- **Extending \texttt{VkRayTracingShaderGroupTypeKHR}:**
  - \texttt{VK\_RAY\_TRACING\_SHADER\_GROUP\_TYPE\_GENERAL\_NV}
• Extending `VkShaderStageFlagBits`:
  ◦ `VK_SHADER_STAGE_ANY_HIT_BIT_NV`
  ◦ `VK_SHADER_STAGE_CALLABLE_BIT_NV`
  ◦ `VK_SHADER_STAGE_CLOSEST_HIT_BIT_NV`
  ◦ `VK_SHADER_STAGE_INTERSECTION_BIT_NV`
  ◦ `VK_SHADER_STAGE_MISS_BIT_NV`
  ◦ `VK_SHADER_STAGE_RAYGEN_BIT_NV`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_CREATE_INFO_NV`
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_INFO_NV`
  ◦ `VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_MEMORY_REQUIREMENTS_INFO_NV`
  ◦ `VK_STRUCTURE_TYPE_BIND_ACCELERATION_STRUCTURE_MEMORY_INFO_NV`
  ◦ `VK_STRUCTURE_TYPE_GEOMETRY_AABB_NV`
  ◦ `VK_STRUCTURE_TYPE_GEOMETRY_NV`
  ◦ `VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_NV`
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PROPERTIES_NV`
  ◦ `VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_NV`
  ◦ `VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_NV`
  ◦ `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_NV`

New or Modified Built-In Variables

• `LaunchIdNV`
• `LaunchSizeNV`
• `WorldRayOriginNV`
• `WorldRayDirectionNV`
• `ObjectRayOriginNV`
• `ObjectRayDirectionNV`
• `RayTminNV`
• `RayTmaxNV`
• `InstanceCustomIndexNV`
• `InstanceId`
• `ObjectToWorldNV`
• `WorldToObjectNV`
New SPIR-V Capabilities

- RayTracingNV

Issues

1) Are there issues?

RESOLVED: Yes.

Sample Code

Example ray generation GLSL shader

```glsl
#version 450 core
#extension GL_NV_ray_tracing : require
layout(set = 0, binding = 0, rgba8) uniform image2D image;
layout(set = 0, binding = 1) uniform accelerationStructureNV as;
layout(location = 0) rayPayloadNV float payload;

void main()
{
    vec4 col = vec4(0, 0, 0, 1);

    vec3 origin = vec3(float(gl_LaunchIDNV.x)/float(gl_LaunchSizeNV.x),
                        float(gl_LaunchIDNV.y)/float(gl_LaunchSizeNV.y), 1.0);
    vec3 dir = vec3(0.0, 0.0, -1.0);

    traceNV(as, 0, 0xff, 0, 1, 0, origin, 0.0, dir, 1000.0, 0);

    col.y = payload;

    imageStore(image, ivec2(gl_LaunchIDNV.xy), col);
}
```

Version History

- Revision 1, 2018-09-11 (Robert Stepinski, Nuno Subtil, Eric Werness)
  - Internal revisions
- Revision 2, 2018-10-19 (Eric Werness)
  - rename to VK_NV_ray_tracing, add support for callables.
VK_NV_representative_fragment_test

Name String
   VK_NV_representative_fragment_test

Extension Type
   Device extension

Registered Extension Number
   167

Revision
   2

Extension and Version Dependencies
   • Requires Vulkan 1.0

Contact
   • Kedarnath Thangudu @kthangudu

Other Extension Metadata

Last Modified Date
   2018-09-13

Contributors
   • Kedarnath Thangudu, NVIDIA
   • Christoph Kubisch, NVIDIA
   • Pierre Boudier, NVIDIA
   • Pat Brown, NVIDIA
   • Jeff Bolz, NVIDIA
   • Eric Werness, NVIDIA

Description

This extension provides a new representative fragment test that allows implementations to reduce the amount of rasterization and fragment processing work performed for each point, line, or triangle primitive. For any primitive that produces one or more fragments that pass all other early fragment tests, the implementation is permitted to choose one or more “representative” fragments for processing and discard all other fragments. For draw calls rendering multiple points, lines, or triangles arranged in lists, strips, or fans, the representative fragment test is performed...
independently for each of those primitives.

This extension is useful for applications that use an early render pass to determine the full set of primitives that would be visible in the final scene. In this render pass, such applications would set up a fragment shader that enables early fragment tests and writes to an image or shader storage buffer to record the ID of the primitive that generated the fragment. Without this extension, the shader would record the ID separately for each visible fragment of each primitive. With this extension, fewer stores will be performed, particularly for large primitives.

The representative fragment test has no effect if early fragment tests are not enabled via the fragment shader. The set of fragments discarded by the representative fragment test is implementation-dependent and may vary from frame to frame. In some cases, the representative fragment test may not discard any fragments for a given primitive.

New Structures

- Extending `VkGraphicsPipelineCreateInfo`:
  - `VkPipelineRepresentativeFragmentTestStateCreateInfoNV`
- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceRepresentativeFragmentTestFeaturesNV`

New Enum Constants

- `VK_NV_REPRESENTATIVE_FRAGMENT_TEST_EXTENSION_NAME`
- `VK_NV_REPRESENTATIVE_FRAGMENT_TEST_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_REPRESENTATIVE_FRAGMENT_TEST_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PIPELINE_REPRESENTATIVE_FRAGMENT_TEST_STATE_CREATE_INFO_NV`

Issues

(1) Is the representative fragment test guaranteed to have any effect?

**RESOLVED**: No. As specified, we only guarantee that each primitive with at least one fragment that passes prior tests will have one fragment passing the representative fragment tests. We don't guarantee that any particular fragment will fail the test.

In the initial implementation of this extension, the representative fragment test is treated as an optimization that may be completely disabled for some pipeline states. This feature was designed for a use case where the fragment shader records information on individual primitives using shader storage buffers or storage images, with no writes to color or depth buffers.

(2) Will the set of fragments that pass the representative fragment test be repeatable if you draw the same scene over and over again?

**RESOLVED**: No. The set of fragments that pass the representative fragment test is implementation-dependent and may vary due to the timing of operations performed by the GPU.
(3) What happens if you enable the representative fragment test with writes to color and/or depth render targets enabled?

**RESOLVED:** If writes to the color or depth buffer are enabled, they will be performed for any fragments that survive the relevant tests. Any fragments that fail the representative fragment test will not update color buffers. For the use cases intended for this feature, we don't expect color or depth writes to be enabled.

(4) How do derivatives and automatic texture level of detail computations work with the representative fragment test enabled?

**RESOLVED:** If a fragment shader uses derivative functions or texture lookups using automatic level of detail computation, derivatives will be computed identically whether or not the representative fragment test is enabled. For the use cases intended for this feature, we don't expect the use of derivatives in the fragment shader.

### Version History

- Revision 2, 2018-09-13 (pbrown)
  - Add issues.
- Revision 1, 2018-08-22 (Kedarnath Thangudu)
  - Internal Revisions

### VK_NV_sample_mask_override_coverage

**Name String**

`VK_NV_sample_mask_override_coverage`

**Extension Type**

Device extension

**Registered Extension Number**

95

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Piers Daniell (pdaniell-nv)

### Other Extension Metadata

**Last Modified Date**

2016-12-08
IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_NV_sample_mask_override_coverage
• This extension provides API support for GL_NV_sample_mask_override_coverage

Contributors
• Daniel Koch, NVIDIA
• Jeff Bolz, NVIDIA

Description
This extension adds support for the following SPIR-V extension in Vulkan:

• SPV_NV_sample_mask_override_coverage

The extension provides access to the OverrideCoverageNV decoration under the SampleMaskOverrideCoverageNV capability. Adding this decoration to a variable with the SampleMask builtin decoration allows the shader to modify the coverage mask and affect which samples are used to process the fragment.

When using GLSL source-based shader languages, the override_coverage layout qualifier from GL_NV_sample_mask_override_coverage maps to the OverrideCoverageNV decoration. To use the override_coverage layout qualifier in GLSL the GL_NV_sample_mask_override_coverage extension must be enabled. Behavior is described in the GL_NV_sample_mask_override_coverage extension spec.

New Enum Constants
• VK_NV_SAMPLE_MASK_OVERRIDE_COVERAGE_EXTENSION_NAME
• VK_NV_SAMPLE_MASK_OVERRIDE_COVERAGE_SPEC_VERSION

New Variable Decoration
• OverrideCoverageNV in SampleMask

New SPIR-V Capabilities
• SampleMaskOverrideCoverageNV

Version History
• Revision 1, 2016-12-08 (Piers Daniell)
  ◦ Internal revisions
VK_NV_scissor_exclusive

Name String

VK_NV_scissor_exclusive

Extension Type

Device extension

Registered Extension Number

206

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact

• Pat Brown @nvpbrown

Other Extension Metadata

Last Modified Date

2018-07-31

IP Status

No known IP claims.

Interactions and External Dependencies

None

Contributors

• Pat Brown, NVIDIA
• Jeff Bolz, NVIDIA
• Piers Daniell, NVIDIA
• Daniel Koch, NVIDIA

Description

This extension adds support for an exclusive scissor test to Vulkan. The exclusive scissor test behaves like the scissor test, except that the exclusive scissor test fails for pixels inside the corresponding rectangle and passes for pixels outside the rectangle. If the same rectangle is used for both the scissor and exclusive scissor tests, the exclusive scissor test will pass if and only if the scissor test fails.
New Commands

- `vkCmdSetExclusiveScissorNV`

New Structures

- Extending `VkPhysicalDeviceFeatures2, VkDeviceCreateInfo`:
  - `VkPhysicalDeviceExclusiveScissorFeaturesNV`
- Extending `VkPipelineViewportStateCreateInfo`:
  - `VkPipelineViewportExclusiveScissorStateCreateInfoNV`

New Enum Constants

- `VK_NV_SCISSOR_EXCLUSIVE_EXTENSION_NAME`
- `VK_NV_SCISSOR_EXCLUSIVE_SPEC_VERSION`
- Extending `VkDynamicState`:
  - `VK_DYNAMIC_STATE_EXCLUSIVE_SCISSOR_NV`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXCLUSIVE_SCISSOR_FEATURES_NV`
  - `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_EXCLUSIVE_SCISSOR_STATE_CREATE_INFO_NV`

Issues

1) For the scissor test, the viewport state must be created with a matching number of scissor and viewport rectangles. Should we have the same requirement for exclusive scissors?

**RESOLVED**: For exclusive scissors, we relax this requirement and allow an exclusive scissor rectangle count that is either zero or equal to the number of viewport rectangles. If you pass in an exclusive scissor count of zero, the exclusive scissor test is treated as disabled.

Version History

- Revision 1, 2018-07-31 (Pat Brown)
  - Internal revisions

**VK_NV_shader_image_footprint**

Name String

`VK_NV_shader_image_footprint`

Extension Type

Device extension

Registered Extension Number

205
Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Contact
• Pat Brown @nvpbrown

Other Extension Metadata

Last Modified Date
2018-09-13

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_NV_shader_image_footprint
• This extension provides API support for GL_NV_shader_texture_footprint

Contributors
• Pat Brown, NVIDIA
• Chris Lentini, NVIDIA
• Daniel Koch, NVIDIA
• Jeff Bolz, NVIDIA

Description
This extension adds Vulkan support for the SPV_NV_shader_image_footprint SPIR-V extension. That SPIR-V extension provides a new instruction OpImageSampleFootprintNV allowing shaders to determine the set of texels that would be accessed by an equivalent filtered texture lookup.

Instead of returning a filtered texture value, the instruction returns a structure that can be interpreted by shader code to determine the footprint of a filtered texture lookup. This structure includes integer values that identify a small neighborhood of texels in the image being accessed and a bitfield that indicates which texels in that neighborhood would be used. The structure also includes a bitfield where each bit identifies whether any texel in a small aligned block of texels would be fetched by the texture lookup. The size of each block is specified by an access granularity provided by the shader. The minimum granularity supported by this extension is 2x2 (for 2D textures) and 2x2x2 (for 3D textures); the maximum granularity is 256x256 (for 2D textures) or 64x32x32 (for 3D textures). Each footprint query returns the footprint from a single texture level. When using minification filters that combine accesses from multiple mipmap levels, shaders must perform separate queries for the two levels accessed (“fine” and “coarse”). The footprint query also returns a flag indicating if the texture lookup would access texels from only one mipmap level or
from two neighboring levels.

This extension should be useful for multi-pass rendering operations that do an initial expensive rendering pass to produce a first image that is then used as a texture for a second pass. If the second pass ends up accessing only portions of the first image (e.g., due to visibility), the work spent rendering the non-accessed portion of the first image was wasted. With this feature, an application can limit this waste using an initial pass over the geometry in the second image that performs a footprint query for each visible pixel to determine the set of pixels that it needs from the first image. This pass would accumulate an aggregate footprint of all visible pixels into a separate “footprint image” using shader atomics. Then, when rendering the first image, the application can kill all shading work for pixels not in this aggregate footprint.

This extension has a number of limitations. The `OpImageSampleFootprintNV` instruction only supports for two- and three-dimensional textures. Footprint evaluation only supports the CLAMP_TO_EDGE wrap mode; results are undefined for all other wrap modes. Only a limited set of granularity values and that set does not support separate coverage information for each texel in the original image.

When using SPIR-V generated from the OpenGL Shading Language, the new instruction will be generated from code using the new `textureFootprint*NV` built-in functions from the `GL_NV_shader_texture_footprint` shading language extension.

**New Structures**

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDeviceShaderImageFootprintFeaturesNV`

**New Enum Constants**

- `VK_NV_SHADER_IMAGE_FOOTPRINT_EXTENSION_NAME`
- `VK_NV_SHADER_IMAGE_FOOTPRINT_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_IMAGE_FOOTPRINT_FEATURES_NV`

**New SPIR-V Capability**

- `ImageFootprintNV`

**Issues**

(1) The footprint returned by the SPIR-V instruction is a structure that includes an anchor, an offset, and a mask that represents a 8x8 or 4x4x4 neighborhood of texel groups. But the bits of the mask are not stored in simple pitch order. Why is the footprint built this way?

**RESOLVED**: We expect that applications using this feature will want to use a fixed granularity and accumulate coverage information from the returned footprints into an aggregate “footprint image” that tracks the portions of an image that would be needed by regular texture filtering. If an application is using a two-dimensional image with 4x4 pixel granularity, we expect that the footprint image will use 64-bit texels where each bit in an 8x8 array of bits corresponds to coverage...
for a 4x4 block in the original image. Texel (0,0) in the footprint image would correspond to texels (0,0) through (31,31) in the original image.

In the usual case, the footprint for a single access will fully contained in a 32x32 aligned region of the original texture, which corresponds to a single 64-bit texel in the footprint image. In that case, the implementation will return an anchor coordinate pointing at the single footprint image texel, an offset vector of (0,0), and a mask whose bits are aligned with the bits in the footprint texel. For this case, the shader can simply atomically OR the mask bits into the contents of the footprint texel to accumulate footprint coverage.

In the worst case, the footprint for a single access spans multiple 32x32 aligned regions and may require updates to four separate footprint image texels. In this case, the implementation will return an anchor coordinate pointing at the lower right footprint image texel and an offset will identify how many “columns” and “rows” of the returned 8x8 mask correspond to footprint texels to the left and above the anchor texel. If the anchor is (2,3), the 64 bits of the returned mask are arranged spatially as follows, where each 4x4 block is assigned a bit number that matches its bit number in the footprint image texels:

```
+-------------------------+-------------------------+-------------------------+-------------------------+
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
| -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- | -- -- -- -- -- -- -- -- |
+-------------------------+-------------------------+-------------------------+-------------------------+
```

To accumulate coverage for each of the four footprint image texels, a shader can AND the returned mask with simple masks derived from the x and y offset values and then atomically OR the updated mask bits into the contents of the corresponding footprint texel.
What should an application do to ensure maximum performance when accumulating footprints into an aggregate footprint image?

**RESOLVED:** We expect that the most common usage of this feature will be to accumulate aggregate footprint coverage, as described in the previous issue. Even if you ignore the anisotropic filtering case where the implementation may return a granularity larger than that requested by the caller, each shader invocation will need to use atomic functions to update up to four footprint image texels for each level of detail accessed. Having each active shader invocation perform multiple atomic operations can be expensive, particularly when neighboring invocations will want to update the same footprint image texels.

Techniques can be used to reduce the number of atomic operations performed when accumulating coverage include:

- Have logic that detects returned footprints where all components of the returned offset vector are zero. In that case, the mask returned by the footprint function is guaranteed to be aligned with the footprint image texels and affects only a single footprint image texel.

- Have fragment shaders communicate using built-in functions from the [VK_NV_shader_subgroup_partitioned](#) extension or other shader subgroup extensions. If you have multiple invocations in a subgroup that need to update the same texel (x,y) in the footprint image, compute an aggregate footprint mask across all invocations in the subgroup updating that texel and have a single invocation perform an atomic operation using that aggregate mask.

- When the returned footprint spans multiple texels in the footprint image, each invocation need to perform four atomic operations. In the previous issue, we had an example that computed separate masks for “topLeft”, “topRight”, “bottomLeft”, and “bottomRight”. When the invocations in a subgroup have good locality, it might be the case the “top left” for some invocations might refer to footprint image texel (10,10), while neighbors might have their “top left” texels at (11,10), (10,11), and (11,11). If you compute separate masks for even/odd x and y values instead of left/right or top/bottom, the “odd/odd” mask for all invocations in the subgroup hold coverage for footprint image texel (11,11), which can be updated by a single atomic operation for the entire subgroup.

**Examples**

TBD
**Version History**

- Revision 2, 2018-09-13 (Pat Brown)
  - Add issue (2) with performance tips.
- Revision 1, 2018-08-12 (Pat Brown)
  - Initial draft

**VK_NV_shader_sm_builtins**

**Name String**

VK_NV_shader_sm_builtins

**Extension Type**

Device extension

**Registered Extension Number**

155

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.1

**Contact**

- Daniel Koch @dgkoch

**Other Extension Metadata**

**Last Modified Date**

2019-05-28

**Interactions and External Dependencies**

- This extension requires SPV_NV_shader_sm_builtins.
- This extension enables GL_NV_shader_sm_builtins for GLSL source languages.

**Contributors**

- Jeff Bolz, NVIDIA
- Eric Werness, NVIDIA

**Description**

This extension provides the ability to determine device-specific properties on NVIDIA GPUs. It provides the number of streaming multiprocessors (SMs), the maximum number of warps (subgroups) that can run on an SM, and shader builtins to enable invocations to identify which SM and warp a shader invocation is executing on.
This extension enables support for the SPIR-V ShaderSMBuiltinsNV capability.

These properties and built-ins should typically only be used for debugging purposes.

**New Structures**

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceShaderSMBuiltinsFeaturesNV
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceShaderSMBuiltinsPropertiesNV

**New Enum Constants**

- VK_NV_SHADER_SM_BUILTINS_EXTENSION_NAME
- VK_NV_SHADER_SM_BUILTINS_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_FEATURES_NV
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADER_SM_BUILTINS_PROPERTIES_NV

**New or Modified Built-In Variables**

- WarpsPerSMNV
- SMCountNV
- WarpIDNV
- SMIDNV

**New SPIR-V Capabilities**

- ShaderSMBuiltinsNV

**Issues**

1. What should we call this extension?

   **RESOLVED**: NV_shader_sm_builtins. Other options considered included:
   - NV_shader_smid - but SMID is really easy to typo/confuse as SIMD.
   - NV_shader_sm_info - but Info is typically reserved for input structures

**Version History**

- Revision 1, 2019-05-28 (Daniel Koch)
  - Internal revisions
**VK_NV_shader_subgroup_partitioned**

**Name String**

VK_NV_shader_subgroup_partitioned

**Extension Type**

Device extension

**Registered Extension Number**

199

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.1

**Contact**

- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

**Other Extension Metadata**

**Last Modified Date**

2018-03-17

**Interactions and External Dependencies**

- This extension requires SPV_NV_shader_subgroup_partitioned
- This extension provides API support for GL_NV_shader_subgroup_partitioned

**Contributors**

- Jeff Bolz, NVIDIA

**Description**

This extension enables support for a new class of group operations on subgroups via the GL_NV_shader_subgroup_partitioned GLSL extension and SPV_NV_shader_subgroup_partitioned SPIR-V extension. Support for these new operations is advertised via the VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV bit.

This extension requires Vulkan 1.1, for general subgroup support.

**New Enum Constants**

- VK_NV_SHADER_SUBGROUP_PARTITIONED_EXTENSION_NAME
- VK_NV_SHADER_SUBGROUP_PARTITIONED_SPEC_VERSION
- Extending VkSubgroupFeatureFlagBits:
  - VK_SUBGROUP_FEATURE_PARTITIONED_BIT_NV
**Version History**

- Revision 1, 2018-03-17 (Jeff Bolz)
  - Internal revisions

**VK_NV_shading_rate_image**

**Name String**

VK_NV_shading_rate_image

**Extension Type**

Device extension

**Registered Extension Number**

165

**Revision**

3

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

**Contact**

- Pat Brown [nvpbrown](https://github.com/nvpbrown)

**Other Extension Metadata**

**Last Modified Date**

2019-07-18

**Interactions and External Dependencies**

- This extension requires SPV_NV_shading_rate
- This extension provides API support for GL_NV_shading_rate_image

**Contributors**

- Pat Brown, NVIDIA
- Carsten Rohde, NVIDIA
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA
- Mathias Schott, NVIDIA
- Matthew Netsch, Qualcomm Technologies, Inc.
**Description**

This extension allows applications to use a variable shading rate when processing fragments of rasterized primitives. By default, Vulkan will spawn one fragment shader for each pixel covered by a primitive. In this extension, applications can bind a *shading rate image* that can be used to vary the number of fragment shader invocations across the framebuffer. Some portions of the screen may be configured to spawn up to 16 fragment shaders for each pixel, while other portions may use a single fragment shader invocation for a 4x4 block of pixels. This can be useful for use cases like eye tracking, where the portion of the framebuffer that the user is looking at directly can be processed at high frequency, while distant corners of the image can be processed at lower frequency. Each texel in the shading rate image represents a fixed-size rectangle in the framebuffer, covering 16x16 pixels in the initial implementation of this extension. When rasterizing a primitive covering one of these rectangles, the Vulkan implementation reads a texel in the bound shading rate image and looks up the fetched value in a palette to determine a base shading rate.

In addition to the API support controlling rasterization, this extension also adds Vulkan support for the *SPV_NV_shading_rate* extension to SPIR-V. That extension provides two fragment shader variable decorations that allow fragment shaders to determine the shading rate used for processing the fragment:

- **FragmentSizeNV**, which indicates the width and height of the set of pixels processed by the fragment shader.
- **InvocationsPerPixel**, which indicates the maximum number of fragment shader invocations that could be spawned for the pixel(s) covered by the fragment.

When using SPIR-V in conjunction with the OpenGL Shading Language (GLSL), the fragment shader capabilities are provided by the *GL_NV_shading_rate_image* language extension and correspond to the built-in variables *gl_FragmentSizeNV* and *gl_InvoctionsPerPixelNV*, respectively.

**New Commands**

- `vkCmdBindShadingRateImageNV`
- `vkCmdSetCoarseSampleOrderNV`
- `vkCmdSetViewportShadingRatePaletteNV`

**New Structures**

- `VkCoarseSampleLocationNV`
- `VkCoarseSampleOrderCustomNV`
- `VkShadingRatePaletteNV`

**Extending** `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:

- `VkPhysicalDeviceShadingRateImageFeaturesNV`

**Extending** `VkPhysicalDeviceProperties2`:

- `VkPhysicalDeviceShadingRateImagePropertiesNV`

**Extending** `VkPipelineViewportStateCreateInfo`:
New Enums

- VkCoarseSampleOrderTypeNV
- VkShadingRatePaletteEntryNV

New Enum Constants

- VK_NV_SHADING_RATE_IMAGE_EXTENSION_NAME
- VK_NV_SHADING_RATE_IMAGE_SPEC_VERSION
- Extending VkAccessFlagBits:
  - VK_ACCESS_SHADING_RATE_IMAGE_READ_BIT_NV
- Extending VkDynamicState:
  - VK_DYNAMIC_STATE_VIEWPORT_COARSE_SAMPLE_ORDER_NV
  - VK_DYNAMIC_STATE_VIEWPORT_SHADING_RATE_PALETTE_NV
- Extending VkImageLayout:
  - VK_IMAGE_LAYOUT_SHADING_RATE_OPTIMAL_NV
- Extending VkImageUsageFlagBits:
- Extending VkPipelineStageFlagBits:
  - VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_FEATURES_NV
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SHADING_RATE_IMAGE_PROPERTIES_NV
  - VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_COARSE_SAMPLE_ORDER_STATE_CREATE_INFO_NV
  - VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SHADING_RATE_IMAGE_STATE_CREATE_INFO_NV

Issues

(1) When using shading rates specifying “coarse” fragments covering multiple pixels, we will generate a combined coverage mask that combines the coverage masks of all pixels covered by the fragment. By default, these masks are combined in an implementation-dependent order. Should we provide a mechanism allowing applications to query or specify an exact order?

**RESOLVED:** Yes, this feature is useful for cases where most of the fragment shader can be evaluated once for an entire coarse fragment, but where some per-pixel computations are also required. For example, a per-pixel alpha test may want to kill all the samples for some pixels in a coarse fragment. This sort of test can be implemented using an output sample mask, but such a shader would need to know which bit in the mask corresponds to each sample in the coarse
fragment. We are including a mechanism to allow applications to specify the orders of coverage samples for each shading rate and sample count, either as static pipeline state or dynamically via a command buffer. This portion of the extension has its own feature bit.

We will not be providing a query to determine the implementation-dependent default ordering. The thinking here is that if an application cares enough about the coarse fragment sample ordering to perform such a query, it could instead just set its own order, also using custom per-pixel sample locations if required.

(2) For the pipeline stage VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV, should we specify a precise location in the pipeline the shading rate image is accessed (after geometry shading, but before the early fragment tests) or leave it under-specified in case there are other implementations that access the image in a different pipeline location?

**RESOLVED** We are specifying the pipeline stage to be between the final stage used for vertex processing (VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT) and before the first stage used for fragment processing (VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT), which seems to be the natural place to access the shading rate image.

(3) How do centroid-sampled variables work with fragments larger than one pixel?

**RESOLVED** For single-pixel fragments, fragment shader inputs decorated with `Centroid` are sampled at an implementation-dependent location in the intersection of the area of the primitive being rasterized and the area of the pixel that corresponds to the fragment. With multi-pixel fragments, we follow a similar pattern, using the intersection of the primitive and the set of pixels corresponding to the fragment.

One important thing to keep in mind when using such “coarse” shading rates is that fragment attributes are sampled at the center of the fragment by default, regardless of the set of pixels/samples covered by the fragment. For fragments with a size of 4x4 pixels, this center location will be more than two pixels (1.5 * sqrt(2)) away from the center of the pixels at the corners of the fragment. When rendering a primitive that covers only a small part of a coarse fragment, sampling a color outside the primitive can produce overly bright or dark color values if the color values have a large gradient. To deal with this, an application can use centroid sampling on attributes where “extrapolation” artifacts can lead to overly bright or dark pixels. Note that this same problem also exists for multisampling with single-pixel fragments, but is less severe because it only affects certain samples of a pixel and such bright/dark samples may be averaged with other samples that don't have a similar problem.

**Version History**

- **Revision 3, 2019-07-18 (Mathias Schott)**
  - Fully list extension interfaces in this appendix.
- **Revision 2, 2018-09-13 (Pat Brown)**
  - Miscellaneous edits preparing the specification for publication.
- **Revision 1, 2018-08-08 (Pat Brown)**
  - Internal revisions
VK_NV_viewport_array2

Name String
   VK_NV_viewport_array2

Extension Type
   Device extension

Registered Extension Number
   97

Revision
   1

Extension and Version Dependencies
   • Requires Vulkan 1.0

Contact
   • Daniel Koch dgkoch

Other Extension Metadata

Last Modified Date
   2017-02-15

Interactions and External Dependencies
   • This extension requires SPV_NV_viewport_array2
   • This extension provides API support for GL_NV_viewport_array2
   • This extension requires the geometryShader and multiViewport features.
   • This extension interacts with the tessellationShader feature.

Contributors
   • Piers Daniell, NVIDIA
   • Jeff Bolz, NVIDIA

Description

This extension adds support for the following SPIR-V extension in Vulkan:

   • SPV_NV_viewport_array2

which allows a single primitive to be broadcast to multiple viewports and/or multiple layers. A new shader built-in output ViewportMaskNV is provided, which allows a single primitive to be output to multiple viewports simultaneously. Also, a new SPIR-V decoration is added to control whether the effective viewport index is added into the variable decorated with the Layer built-in decoration. These capabilities allow a single primitive to be output to multiple layers simultaneously.
This extension allows variables decorated with the `Layer` and `ViewportIndex` built-ins to be exported from vertex or tessellation shaders, using the `ShaderViewportIndexLayerNV` capability.

This extension adds a new `ViewportMaskNV` built-in decoration that is available for output variables in vertex, tessellation evaluation, and geometry shaders, and a new `ViewportRelativeNV` decoration that can be added on variables decorated with `Layer` when using the `ShaderViewportMaskNV` capability.

When using GLSL source-based shading languages, the `gl_ViewportMask[]` built-in output variable and `viewport_relative` layout qualifier from `GL_NV_viewport_array2` map to the `ViewportMaskNV` and `ViewportRelativeNV` decorations, respectively. Behaviour is described in the `GL_NV_viewport_array2` extension specification.

```
Note
The `ShaderViewportIndexLayerNV` capability is equivalent to the `ShaderViewportIndexLayerEXT` capability added by `VK_EXT_shader_viewport_index_layer`.
```

New Enum Constants

- `VK_NV_VIEWPORT_ARRAY2_EXTENSION_NAME`
- `VK_NV_VIEWPORT_ARRAY2_SPEC_VERSION`

New or Modified Built-In Variables

- (modified) `Layer`
- (modified) `ViewportIndex`
- `ViewportMaskNV`

New Variable Decoration

- `ViewportRelativeNV` in `Layer`

New SPIR-V Capabilities

- `ShaderViewportIndexLayerNV`
- `ShaderViewportMaskNV`

Version History

- Revision 1, 2017-02-15 (Daniel Koch)
  - Internal revisions

`VK_NV_viewport_swizzle`
Name String

VK_NV_viewport_swizzle

Extension Type

Device extension

Registered Extension Number

99

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0

Contact

• Piers Daniell @pdaniell-nv

Other Extension Metadata

Last Modified Date

2016-12-22

Interactions and External Dependencies

• This extension requires multiViewport and geometryShader features to be useful.

Contributors

• Daniel Koch, NVIDIA
• Jeff Bolz, NVIDIA

Description

This extension provides a new per-viewport swizzle that can modify the position of primitives sent to each viewport. New viewport swizzle state is added for each viewport, and a new position vector is computed for each vertex by selecting from and optionally negating any of the four components of the original position vector.

This new viewport swizzle is useful for a number of algorithms, including single-pass cubemap rendering (broadcasting a primitive to multiple faces and reorienting the vertex position for each face) and voxel rasterization. The per-viewport component remapping and negation provided by the swizzle allows application code to re-orient three-dimensional geometry with a view along any of the X, Y, or Z axes. If a perspective projection and depth buffering is required, 1/W buffering should be used, as described in the single-pass cubemap rendering example in the “Issues” section below.

New Structures

• VkViewportSwizzleNV
• Extending `VkPipelineViewportStateCreateInfo`:
  ◦ `VkPipelineViewportSwizzleStateCreateInfoNV`

**New Enums**

• `VkViewportCoordinateSwizzleNV`

**New Bitmasks**

• `VkPipelineViewportSwizzleStateCreateFlagsNV`

**New Enum Constants**

• `VK_NV_VIEWPORT_SWIZZLE_EXTENSION_NAME`
• `VK_NV_VIEWPORT_SWIZZLE_SPEC_VERSION`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_SWIZZLE_STATE_CREATE_INFO_NV`

**Issues**

1) Where does viewport swizzling occur in the pipeline?

**RESOLVED:** Despite being associated with the viewport, viewport swizzling must happen prior to the viewport transform. In particular, it needs to be performed before clipping and perspective division.

The viewport mask expansion (`VK_NV_viewport_array2`) and the viewport swizzle could potentially be performed before or after transform feedback, but feeding back several viewports worth of primitives with different swizzles doesn’t seem particularly useful. This specification applies the viewport mask and swizzle after transform feedback, and makes primitive queries only count each primitive once.

2) Any interesting examples of how this extension, `VK_NV_viewport_array2`, and `VK_NV_geometry_shader_passthrough` can be used together in practice?

**RESOLVED:** One interesting use case for this extension is for single-pass rendering to a cubemap. In this example, the application would attach a cubemap texture to a layered FBO where the six cube faces are treated as layers. Vertices are sent through the vertex shader without applying a projection matrix, where the `gl_Position` output is (x,y,z,1) and the center of the cubemap is at (0,0,0). With unextended Vulkan, one could have a conventional instanced geometry shader that looks something like the following:
layout(invocations = 6) in;  // separate invocation per face
layout(triangles) in;
layout(triangle_strip) out;
layout(max_vertices = 3) out;

in Inputs {
  vec2 texcoord;
  vec3 normal;
  vec4 baseColor;
} v[];

out Outputs {
  vec2 texcoord;
  vec3 normal;
  vec4 baseColor;
};

void main()
{
  int face = gl_InvocationID;  // which face am I?

  // Project gl_Position for each vertex onto the cube map face.
  vec4 positions[3];
  for (int i = 0; i < 3; i++) {
    positions[i] = rotate(gl_in[i].gl_Position, face);
  }

  // If the primitive doesn't project onto this face, we're done.
  if (shouldCull(positions)) {
    return;
  }

  // Otherwise, emit a copy of the input primitive to the
  // appropriate face (using gl_Layer).
  for (int i = 0; i < 3; i++) {
    gl_Layer = face;
    gl_Position = positions[i];
    texcoord = v[i].texcoord;
    normal = v[i].normal;
    baseColor = v[i].baseColor;
    EmitVertex();
  }
}

With passthrough geometry shaders, this can be done using a much simpler shader:
layout(triangles) in;
layout(passthrough) in Inputs {
  vec2 texcoord;
  vec3 normal;
  vec4 baseColor;
}
layout(passthrough) in gl_PerVertex {
  vec4 gl_Position;
} gl_in[];
layout(viewport_relative) out int gl_Layer;

void main()
{
  // Figure out which faces the primitive projects onto and
  // generate a corresponding viewport mask.
  uint mask = 0;
  for (int i = 0; i < 6; i++) {
    if (!shouldCull(face)) {
      mask |= 1U << i;
    }
  }
  gl_ViewportMask = mask;
  gl_Layer = 0;
}

The application code is set up so that each of the six cube faces has a separate viewport (numbered 0 to 5). Each face also has a separate swizzle, programmed via the VkPipelineViewportSwizzleStateCreateInfoNV pipeline state. The viewport swizzle feature performs the coordinate transformation handled by the rotate() function in the original shader. The viewport_relative layout qualifier says that the viewport number (0 to 5) is added to the base gl_Layer value of 0 to determine which layer (cube face) the primitive should be sent to.

Note that the use of the passed through input normal in this example suggests that the fragment shader in this example would perform an operation like per-fragment lighting. The viewport swizzle would transform the position to be face-relative, but normal would remain in the original coordinate system. It seems likely that the fragment shader in either version of the example would want to perform lighting in the original coordinate system. It would likely do this by reconstructing the position of the fragment in the original coordinate system using gl_FragCoord, a constant or uniform holding the size of the cube face, and the input gl_ViewportIndex (or gl_Layer), which identifies the cube face. Since the value of normal is in the original coordinate system, it would not need to be modified as part of this coordinate transformation.

Note that while the rotate() operation in the regular geometry shader above could include an arbitrary post-rotation projection matrix, the viewport swizzle does not support arbitrary math. To get proper projection, 1/W buffering should be used. To do this:

1. Program the viewport swizzles to move the pre-projection W eye coordinate (typically 1.0) into the Z coordinate of the swizzle output and the eye coordinate component used for depth into the W coordinate. For example, the viewport corresponding to the +Z face might use a swizzle of
(+X, -Y, +W, +Z). The Z normalized device coordinate computed after swizzling would then be 
\[ z'/w' = \frac{1}{Z_{\text{eye}}} \].

2. On NVIDIA implementations supporting floating-point depth buffers with values outside [0,1], prevent unwanted near plane clipping by enabling `depthClampEnable`. Ensure that the depth clamp doesn’t mess up depth testing by programming the depth range to very large values, such as \( \text{minDepthBounds}=z, \text{maxDepthBounds}=+z \), where \( z = 2^{127} \). It should be possible to use IEEE infinity encodings also (0x7F800000 for -INF, 0xFF800000 for +INF). Even when near/far clipping is disabled, primitives extending behind the eye will still be clipped because one or more vertices will have a negative W coordinate and fail X/Y clipping tests.

On other implementations, scale X, Y, and Z eye coordinates so that vertices on the near plane have a post-swizzle W coordinate of 1.0. For example, if the near plane is at \( Z_{\text{eye}} = 1/256 \), scale X, Y, and Z by 256.

3. Adjust depth testing to reflect the fact that 1/W values are large near the eye and small away from the eye. Clear the depth buffer to zero (infinitely far away) and use a depth test of `VK_COMPARE_OP_GREATER` instead of `VK_COMPARE_OP_LESS`.

**Version History**

- Revision 1, 2016-12-22 (Piers Daniell)
  - Internal revisions

**VK_NVX_image_view_handle**

**Name String**

\[ \text{VK_NVX_image_view_handle} \]

**Extension Type**

Device extension

**Registered Extension Number**

31

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Contact**

- Eric Werness [ewerness](mailto:ewerness@)
Contributors
- Eric Werness, NVIDIA
- Jeff Bolz, NVIDIA
- Daniel Koch, NVIDIA

Description
This extension allows applications to query an opaque handle from an image view for use as a sampled image or storage image. This provides no direct functionality itself.

New Commands
- vkGetImageViewAddressNVX
- vkGetImageViewHandleNVX

New Structures
- VkImageViewAddressPropertiesNVX
- VkImageViewHandleInfoNVX

New Enum Constants
- VK_NVX_IMAGE_VIEW_HANDLE_EXTENSION_NAME
- VK_NVX_IMAGE_VIEW_HANDLE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IMAGE_VIEW_ADDRESS_PROPERTIES_NVX
  - VK_STRUCTURE_TYPE_IMAGE_VIEW_HANDLE_INFO_NVX

Version History
- Revision 2, 2020-04-03 (Piers Daniell)
  - Add vkGetImageViewAddressNVX
- Revision 1, 2018-12-07 (Eric Werness)
  - Internal revisions

VK_NVX_multiview_per_view_attributes

Name String
VK_NVX_multiview_per_view_attributes

Extension Type
Device extension
This extension adds a new way to write shaders to be used with multiview subpasses, where the attributes for all views are written out by a single invocation of the vertex processing stages. Related SPIR-V and GLSL extensions `SPV_NVX_multiview_per_view_attributes` and `GL_NVX_multiview_per_view_attributes` introduce per-view position and viewport mask attributes arrays, and this extension defines how those per-view attribute arrays are interpreted by Vulkan. Pipelines using per-view attributes may only execute the vertex processing stages once for all views rather than once per-view, which reduces redundant shading work.

A subpass creation flag controls whether the subpass uses this extension. A subpass must either exclusively use this extension or not use it at all.

Some Vulkan implementations only support the position attribute varying between views in the X component. A subpass can declare via a second creation flag whether all pipelines compiled for this subpass will obey this restriction.
Shaders that use the new per-view outputs (e.g. `gl_PositionPerViewNV`) must also write the non-per-view output (`gl_Position`), and the values written must be such that `gl_Position = gl_PositionPerViewNV[gl_ViewIndex]` for all views in the subpass. Implementations are free to either use the per-view outputs or the non-per-view outputs, whichever would be more efficient.

If `VK_NV_viewport_array2` is not also supported and enabled, the per-view viewport mask must not be used.

**New Structures**

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceMultiviewPerViewAttributesPropertiesNVX`

**New Enum Constants**

- `VK_NVX_MULTIVIEW_PER_VIEW_ATTRIBUTES_EXTENSION_NAME`
- `VK_NVX Multiview Per View Attributes Spec Version`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PER_VIEW_ATTRIBUTES_PROPERTIES_NVX`
- Extending `VkSubpassDescriptionFlagBits`:
  - `VK_SUBPASS_DESCRIPTION_PER_VIEW_ATTRIBUTES_BIT_NVX`
  - `VK_SUBPASS_DESCRIPTION_PER_VIEW_POSITION_X_ONLY_BIT_NVX`

**New Built-In Variables**

- `PositionPerViewNV`
- `ViewportMaskPerViewNV`

**New SPIR-V Capabilities**

- `PerViewAttributesNV`

**Examples**
#version 450 core

#extension GL_KHX_multiview : enable
#extension GL_NVX_multiview_per_view_attributes : enable

layout(location = 0) in vec4 position;
layout(set = 0, binding = 0) uniform Block { mat4 mvpPerView[2]; } buf;

void main()
{
    // Output both per-view positions and gl_Position as a function of gl_ViewIndex
    gl_PositionPerViewNV[0] = buf.mvpPerView[0] * position;
    gl_Position = buf.mvpPerView[gl_ViewIndex] * position;
}

Version History

- Revision 1, 2017-01-13 (Jeff Bolz)
  - Internal revisions

VK_QCOM_render_pass_shader_resolve

Name String

VK_QCOM_render_pass_shader_resolve

Extension Type

Device extension

Registered Extension Number

172

Revision

4

Extension and Version Dependencies

- Requires Vulkan 1.0

Contact

- Bill Licea-Kane

Other Extension Metadata

Last Modified Date

2019-11-07
IP Status
No known IP claims.

Interactions and External Dependencies
None.

Contributors
• Srihari Babu Alla, Qualcomm
• Bill Licea-Kane, Qualcomm
• Jeff Leger, Qualcomm

Description
This extension allows a shader resolve to replace fixed-function resolve.

Fixed-function resolve is limited in function to simple filters of multisample buffers to a single sample buffer.

Fixed-function resolve is more performance efficient and/or power efficient than shader resolve for such simple filters.

Shader resolve allows a shader writer to create complex, non-linear filtering of a multisample buffer in the last subpass of a subpass dependency chain.

This extension also provides a bit which can be used to enlarge a sample region dependency to a fragment region dependency, so that a framebuffer-region dependency can replace a framebuffer-global dependency in some cases.

New Enum Constants
• VK_QCOM_RENDER_PASS_SHADER_RESOLVE_EXTENSION_NAME
• VK_QCOM_RENDER_PASS_SHADER_RESOLVE_SPEC_VERSION
• Extending VkSubpassDescriptionFlagBits:
  ◦ VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM
  ◦ VK_SUBPASS_DESCRIPTION_SHADER_RESOLVE_BIT_QCOM

Issues
1) Should this extension be named render_pass_shader_resolve?

RESOLVED Yes.

This is part of suite of small extensions to render pass.

Following the style guide, instead of following VK_KHR_create_renderpass2.

2) Should the VK_SAMPLE_COUNT_1_BIT be required for each pColorAttachment and the
DepthStencilAttachment?

**RESOLVED No.**

While this may not be a common use case, and while most fixed-function resolve hardware has this limitation, there is little reason to require a shader resolve to resolve to a single sample buffer.

3) Should a shader resolve subpass be the last subpass in a renderpass?

**RESOLVED Yes.**

To be more specific, it should be the last subpass in a subpass dependency chain.

4) Do we need the `VK_SUBPASS_DESCRIPTION_FRAGMENT_REGION_BIT_QCOM` bit?

**RESOLVED Yes.**

This applies when an input attachment's sample count is equal to `rasterizationSamples`. Further, if `sampleShading` is enabled (explicitly or implicitly) then `minSampleShading` **must** equal 0.0.

However, this bit may be set on any subpass, it is not restricted to a shader resolve subpass.

**Version History**

- Revision 1, 2019-06-28 (wwlk)
  - Initial draft
- Revision 2, 2019-11-06 (wwlk)
  - General clean-up/spec updates
  - Added issues
- Revision 3, 2019-11-07 (wwlk)
  - Typos
  - Additional issues
  - Clarified that a shader resolve subpass is the last subpass in a subpass dependency chain
- Revision 4, 2020-01-06 (wwlk)
  - Change resolution of Issue 1 (`render_pass`, not `renderpass`)

**VK_QCOM_render_pass_store_ops**

**Name String**

`VK_QCOM_render_pass_store_ops`

**Extension Type**

Device extension

**Registered Extension Number**

302
Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0

Contact
• Bill Licea-Kane 🌀wwlk

Other Extension Metadata

Last Modified Date
2020-03-25

Contributors
• Bill Licea-Kane, Qualcomm Technologies, Inc.

Description

Renderpass attachments can be read-only for the duration of a renderpass.

Examples include input attachments and depth attachments where depth tests are enabled but depth writes are not enabled.

In such cases, there can be no contents generated for an attachment within the render area.

This extension adds a new VkAttachmentStoreOp VK_ATTACHMENT_STORE_OP_NONE_QCOM which specifies that the contents within the render area may not be written to memory, but that the prior contents of the attachment in memory are preserved. However, if any contents were generated within the render area during rendering, the contents of the attachment will be undefined inside the render area.

Note
The VkAttachmentStoreOp VK_ATTACHMENT_STORE_OP_STORE may force an implementation to assume that the attachment was written and force an implementation to flush data to memory or to a higher level cache. The VkAttachmentStoreOp VK_ATTACHMENT_STORE_OP_NONE_QCOM may allow an implementation to assume that the attachment was not written and allow an implementation to avoid such a flush.

New Enum Constants

• VK_QCOM_RENDER_PASS_STORE_OPS_EXTENSION_NAME

• VK_QCOM_RENDER_PASS_STORE_OPS_SPEC_VERSION

• Extending VkAttachmentStoreOp:
  ◦ VK_ATTACHMENT_STORE_OP_NONE_QCOM
**Version History**

- Revision 1, 2019-12-20 (wwlk)
  - Initial version
- Revision 2, 2020-03-25 (wwlk)
  - Minor renaming

**VK_QCOM_render_pass_transform**

**Name String**

VK_QCOM_render_pass_transform

**Extension Type**

Device extension

**Registered Extension Number**

283

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_swapchain
- Requires VK_KHR_surface

**Contact**

- Jeff Leger @jackohound

**Other Extension Metadata**

**Last Modified Date**

2021-03-09

**Interactions and External Dependencies**

- This extension requires VK_KHR_swapchain
- This extension interacts with VK_EXT_fragment_density_map
- This extension interacts with VK_KHR_fragment_shading_rate

**Contributors**

- Jeff Leger, Qualcomm Technologies, Inc.
- Brandon Light, Qualcomm Technologies, Inc.
- Matthew Netsch, Qualcomm Technologies, Inc.
This extension provides a mechanism for applications to enable driver support for render pass transform.

Mobile devices can be rotated and mobile applications need to render properly when a device is held in a landscape or portrait orientation. When the current orientation differs from the device’s native orientation, a rotation is required so that the “up” direction of the rendered scene matches the current orientation.

If the Display Processing Unit (DPU) doesn’t natively support rotation, the Vulkan presentation engine can handle this rotation in a separate composition pass. Alternatively, the application can render frames “pre-rotated” to avoid this extra pass. The latter is preferred to reduce power consumption and achieve the best performance because it avoids tasking the GPU with extra work to perform the copy/rotate operation.

Unlike OpenGL ES, the burden of pre-rotation in Vulkan falls on the application. To implement pre-rotation, applications render into swapchain images matching the device native aspect ratio of the display and “pre-rotate” the rendering content to match the device’s current orientation. The burden is more than adjusting the Model View Projection (MVP) matrix in the vertex shader to account for rotation and aspect ratio. The coordinate systems of scissors, viewports, derivatives and several shader built-ins may need to be adapted to produce the correct result.

It is difficult for some game engines to manage this burden; many chose to simply accept the performance/power overhead of performing rotation in the presentation engine.

This extension allows applications to achieve the performance benefits of pre-rotated rendering by moving much of the above-mentioned burden to the graphics driver. The following is unchanged with this extension:

- Applications create a swapchain matching the native orientation of the display. Applications must also set the `VkSwapchainCreateInfoKHR::preTransform` equal to the `currentTransform` as returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`.

The following is changed with this extension:

- At `vkCmdBeginRenderPass`, the application provides extension struct `VkRenderPassTransformBeginInfoQCOM` specifying the render pass transform parameters.
- At `vkBeginCommandBuffer` for secondary command buffers, the application provides extension struct `VkCommandBufferInheritanceRenderPassTransformInfoQCOM` specifying the render pass transform parameters.
- The `renderArea`, viewports, scissors, and `fragmentSize` are all provided in the current (non-rotated) coordinate system. The implementation will transform those into the native (rotated) coordinate system.
- The implementation is responsible for transforming shader built-ins (`FragCoord`, `PointCoord`, `SamplePosition`, `PrimitiveShadingRateKHR`, `interpolateAt()`, `dFdx`, `dFdy`, `fWidth`) into the rotated coordinate system.
- The implementation is responsible for transforming `position` to the rotated coordinate system.
New Structures

- Extending `VkCommandBufferInheritanceInfo`:
  - `VkCommandBufferInheritanceRenderPassTransformInfoQCOM`

- Extending `VkRenderPassBeginInfo`:
  - `VkRenderPassTransformBeginInfoQCOM`

New Enum Constants

- `VK_QCOM_RENDER_PASS_TRANSFORM_EXTENSION_NAME`
- `VK_QCOM_RENDER_PASS_TRANSFORM_SPEC_VERSION`

- Extending `VkRenderPassCreateFlagBits`:
  - `VK_RENDER_PASS_CREATE_TRANSFORM_BIT_QCOM`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_RENDER_PASS_TRANSFORM_INFO_QCOM`
  - `VK_STRUCTURE_TYPE_RENDER_PASS_TRANSFORM_BEGIN_INFO_QCOM`

Issues

1) Some early Adreno drivers (October 2019 through March 2020) advertised support for this extension but expected VK_STRUCTURE_TYPE values different from those in the vulkan headers. To cover all Adreno devices on the market, applications need to detect the driver version and use the appropriate VK_STRUCTURE_TYPE values from the table below.

The driver version reported in `VkPhysicalDeviceProperties.driverVersion` is a `uint32_t` type. You can decode the `uint32_t` value into a major.minor.patch version as shown below:

```c
uint32_t major = ((driverVersion) >> 22);
uint32_t minor = ((driverVersion) >> 12) & 0x3ff);
uint32_t patch = ((driverVersion) & 0xfff);
```

If the Adreno major.minor.patch version is greater than or equal to 512.469.0, then simply use the VK_STRUCTURE_TYPE values as defined in vulkan_core.h. If the version is less than or equal to 512.468.0, then use the alternate values for the two VK_STRUCTURE_TYPEs in the table below.

<table>
<thead>
<tr>
<th>Adreno Driver Version</th>
<th>512.468.0 and earlier</th>
<th>512.469.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_STRUCTURE_TYPE_RENDER_PASS_TRANSFORM_BEGIN_INFO_QCOM</code></td>
<td>1000282000</td>
<td>1000282001</td>
</tr>
</tbody>
</table>

Table 93. Adreno Driver Requirements
2) Should the extension support only rotations (e.g. 90, 180, 270-degrees), or also mirror transforms (e.g. vertical flips)? Mobile use-cases only require rotation. Other display systems such as projectors might require a flipped transform.

**RESOLVED:** In this version of the extension, the functionality is restricted to 90, 180, and 270-degree rotations to address mobile use-cases.

3) How does this extension interact with VK_EXT_fragment_density_map?

**RESOLVED** Some implementations may not be able to support a render pass that enables both render pass transform and fragment density maps. For simplicity, this extension disallows enabling both features within a single render pass.

4) What should this extension be named?

We considered names such as “rotated_rendering”, “pre_rotation” and others. Since the functionality is limited to a render pass, it seemed the name should include “render_pass”. While the current extension is limited to rotations, it could be extended to other transforms (like mirror) in the future.

**RESOLVED** The name “render_pass_transform” seems like the most accurate description of the introduced functionality.

5) How does this extension interact with VK_KHR_fragment_shading_rate?

**RESOLVED:** For the same reasons as issue 3, this extension disallows enabling both pFragmentShadingRateAttachment and render pass transform within a single render pass.

However, pipeline shading rate and primitive shading rate are supported, and their respective fragmentSize and PrimitiveShadingRateKHR are provided in the current (non-rotated) coordinate system. The implementation is responsible for transforming them to the rotated coordinate system.

The set of supported shading rates may be different per transform. Supported rates queried from vkGetPhysicalDeviceFragmentShadingRatesKHR are in the native (rotated) coordinate system. This means that the application must swap the x/y of the reported rates to get the set of rates supported for 90 and 270 degree rotation.

**Version History**

- Revision 1, 2020-02-05 (Jeff Leger)
- Revision 2, 2021-03-09 (Matthew Netsch)
  - Adds interactions with VK_KHR_fragment_shading_rate
VK_QCOM_rotated_copy_commands

Name String
VK_QCOM_rotated_copy_commands

Extension Type
Device extension

Registered Extension Number
334

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_swapchain
• Requires VK_KHR_copy_commands2

Contact
• Jeff Leger jackohound

Other Extension Metadata

Last Modified Date
2020-09-18

Interactions and External Dependencies
• None

Contributors
• Jeff Leger, Qualcomm Technologies, Inc.

Description

This extension extends adds an optional rotation transform to copy commands vkCmdBlitImage2KHR, vkCmdCopyImageToBuffer2KHR and vkCmdCopyBufferToImage2KHR. When copying between two resources, where one resource contains rotated content and the other does not, a rotated copy may be desired. This extension may be used in combination with VK_QCOM_render_pass_transform which adds rotated renderpasses.

This extension adds an extension structure to the following commands: vkCmdBlitImage2KHR, vkCmdCopyImageToBuffer2KHR and vkCmdCopyBufferToImage2KHR

Issues

1) What is an appropriate name for the added extension structure? The style guide says “Structures which extend other structures through the pNext chain should reflect the name of the base
structure they extend,” but in this case a single extension structure is used to extend three base structures (vkCmdBlitImage2KHR, vkCmdCopyImageToBuffer2KHR and vkCmdCopyBufferToImage2KHR). Creating three identical structures with unique names seemed undesirable.

**RESOLVED**: Deviate from the style guide for extension structure naming.

2) Should this extension add a rotation capability to vkCmdCopyImage2KHR?

**RESOLVED**: No. Use of rotated vkCmdBlitImage2KHR can fully address this use-case.

3) Should this extension add a rotation capability to vkCmdResolveImage2KHR?

**RESOLVED** No. Use of vkCmdResolveImage2KHR is very slow and extremely bandwidth intensive on Qualcomm’s GPU architecture and use of pResolveAttachments in vkRenderPass is the strongly preferred approach. Therefore, we choose not to introduce a rotation capability to vkCmdResolveImage2KHR.

**New Structures**

- Extending `VkBufferImageCopy2KHR`, `VkImageBlit2KHR`:
  - `VkCopyCommandTransformInfoQCOM`

**New Enum Constants**

- `VK_QCOM_ROTATED_COPY_COMMANDS_EXTENSION_NAME`
- `VK_QCOM_ROTATED_COPY_COMMANDS_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_COPY_COMMAND_TRANSFORM_INFO_QCOM`

**Version History**

- Revision 1, 2020-09-19 (Jeff Leger)

**VK_QNX_screen_surface**

**Name String**
- `VK_QNX_screen_surface`

**Extension Type**
- Instance extension

**Registered Extension Number**
- 379

**Revision**
- 1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_surface

Contact

- Mike Gorchak @mgorchak-blackberry

Other Extension Metadata

Last Modified Date
2021-01-11

IP Status
No known IP claims.

Contributors
- Mike Gorchak, BlackBerry Limited

Description

The VK_QNX_screen_surface extension is an instance extension. It provides a mechanism to create a VkSurfaceKHR object (defined by the VK_KHR_surface extension) that refers to a QNX Screen window, as well as a query to determine support for rendering to a QNX Screen compositor.

New Commands

- vkCreateScreenSurfaceQNX
- vkGetPhysicalDeviceScreenPresentationSupportQNX

New Structures

- VkScreenSurfaceCreateInfoQNX

New Bitmasks

- VkScreenSurfaceCreateFlagsQNX

New Enum Constants

- VK_QNX_SCREEN_SURFACE_EXTENSION_NAME
- VK_QNX_SCREEN_SURFACE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_SCREEN_SURFACE_CREATE_INFO_QNX
Version History
• Revision 1, 2021-01-11 (Mike Gorchak)
  ◦ Initial draft.

VK_VALVE.mutable_descriptor_type

Name String
VK_VALVE.mutable_descriptor_type

Extension Type
Device extension

Registered Extension Number
352

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_maintenance3

Special Use
• D3D support

Contact
• Joshua Ashton Joshua-Ashton
• Hans-Kristian Arntzen HansKristian-Work

Other Extension Metadata

Last Modified Date
2020-12-02

IP Status
No known IP claims.

Contributors
• Joshua Ashton, Valve
• Hans-Kristian Arntzen, Valve

Description
This extension allows applications to reduce descriptor memory footprint by allowing a descriptor to be able to mutate to a given list of descriptor types depending on which descriptor types are written into, or copied into a descriptor set.
The main use case this extension intends to address is descriptor indexing with
VK_DESCRIPTOR_BINDING_VARIABLE_DESCRIPTOR_COUNT_BIT where the descriptor types are completely
generic, as this means applications can allocate one large descriptor set, rather than having one
large descriptor set per descriptor type, which significantly bloats descriptor memory usage and
causes performance issues.

This extension also adds a mechanism to declare that a descriptor pool, and therefore the
descriptor sets that are allocated from it, reside only in host memory; as such these descriptors can
only be updated/copied, but not bound.

These features together allow much more efficient emulation of the raw D3D12 binding model. This
extension is primarily intended to be useful for API layering efforts.

New Structures

• VkMutableDescriptorTypeListVALVE

• Extending VkDescriptorSetLayoutCreateInfo, VkDescriptorPoolCreateInfo:
  ◦ VkMutableDescriptorTypeCreateInfoVALVE

• Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  ◦ VkPhysicalDeviceMutableDescriptorTypeFeaturesVALVE

New Enum Constants

• VK_VALVE_MUTABLE_DESCRIPTOR_TYPE_EXTENSION_NAME

• VK_VALVE_MUTABLE_DESCRIPTOR_TYPE_SPEC_VERSION

• Extending VkDescriptorPoolCreateFlagBits:
  ◦ VK_DESCRIPTOR_POOL_CREATE_HOST_ONLY_BIT_VALVE

• Extending VkDescriptorSetLayoutCreateFlagBits:
  ◦ VK_DESCRIPTOR_SET_LAYOUT_CREATE_HOST_ONLY_POOL_BIT_VALVE

• Extending VkDescriptorType:
  ◦ VK_DESCRIPTOR_TYPE_MUTABLE_VALVE

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_MUTABLE_DESCRIPTOR_TYPE_CREATE_INFO_VALVE
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MUTABLE_DESCRIPTOR_TYPE_FEATURES_VALVE

Version History

• Revision 1, 2020-12-01 (Joshua Ashton, Hans-Kristian Arntzen)
  ◦ Initial specification, squashed from public draft.
List of Provisional Extensions

- VK_KHR_portability_subset
- VK_KHR_video_decode_queue
- VK_KHR_video_encode_queue
- VK_KHR_video_queue
- VK_EXT_video_decode_h264
- VK_EXT_video_decode_h265
- VK_EXT_video_encode_h264
VK_KHR_portability_subset

Name String
    VK_KHR_portability_subset

Extension Type
    Device extension

Registered Extension Number
    164

Revision
    1

Extension and Version Dependencies
    • Requires Vulkan 1.0
    • Requires VK_KHR_get_physical_device_properties2
    • This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

Contact
    • Bill Hollings billhollings

Last Modified Date
    2020-07-21

IP Status
    No known IP claims.

Contributors
    • Bill Hollings, The Brenwill Workshop Ltd.
    • Daniel Koch, NVIDIA
    • Dzmitry Malyshau, Mozilla
    • Chip Davis, CodeWeavers
    • Dan Ginsburg, Valve
    • Mike Weiblen, Valve
    • Neil Trevett, NVIDIA

Description
The `VK_KHR_portability_subset` extension allows a non-conformant Vulkan implementation to be built on top of another non-Vulkan graphics API, and identifies differences between that implementation and a fully-conformant native Vulkan implementation.

This extension provides Vulkan implementations with the ability to mark otherwise-required
capabilities as unsupported, or to establish additional properties and limits that the application should adhere to in order to guarantee portable behaviour and operation across platforms, including platforms where Vulkan is not natively supported.

The goal of this specification is to document, and make queryable, capabilities which are required to be supported by a fully-conformant Vulkan 1.0 implementation, but may be optional for an implementation of the Vulkan 1.0 Portability Subset.

The intent is that this extension will be advertised only on implementations of the Vulkan 1.0 Portability Subset, and not on conformant implementations of Vulkan 1.0. Fully-conformant Vulkan implementations provide all the required capabilities, and so will not provide this extension. Therefore, the existence of this extension can be used to determine that an implementation is likely not fully conformant with the Vulkan spec.

If this extension is supported by the Vulkan implementation, the application must enable this extension.

This extension defines several new structures that can be chained to the existing structures used by certain standard Vulkan calls, in order to query for non-conformant portable behavior.

New Structures

- Extending `VkPhysicalDeviceFeatures2`, `VkDeviceCreateInfo`:
  - `VkPhysicalDevicePortabilitySubsetFeaturesKHR`

- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDevicePortabilitySubsetPropertiesKHR`

New Enum Constants

- `VK_KHR_PORTABILITY_SUBSET_EXTENSION_NAME`
- `VK_KHR_PORTABILITY_SUBSET_SPEC_VERSION`

- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PORTABILITY_SUBSET_FEATURES_KHR`
  - `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PORTABILITY_SUBSET_PROPERTIES_KHR`

Issues

None.

Version History

- Revision 1, 2020-07-21 (Bill Hollings)
  - Initial draft.
VK_KHR_video_decode_queue

Name String
  VK_KHR_video_decode_queue

Extension Type
  Device extension

Registered Extension Number
  25

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_video_queue
  • Requires VK_KHR_synchronization2
  • This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

Contact
  • jake.beju@amd.com

Other Extension Metadata

Last Modified Date
  2021-03-29

IP Status
  No known IP claims.

Contributors
  • Jake Beju, AMD
  • Olivier Lapicque, NVIDIA
  • Peter Fang, AMD
  • Piers Daniell, NVIDIA
  • Srinath Kumarapuram, NVIDIA
  • Tony Zlatinski, NVIDIA

New Commands
  • vkCmdDecodeVideoKHR
New Structures

- VkVideoDecodeInfoKHR

New Enums

- VkVideoDecodeFlagBitsKHR

New Bitmasks

- VkVideoDecodeFlagsKHR

New Enum Constants

- VK_KHR_VIDEO_DECODE_QUEUE_EXTENSION_NAME
- VK_KHR_VIDEO_DECODE_QUEUE_SPEC_VERSION
- Extending VkAccessFlagBits2KHR:
  - VK_ACCESS_2_VIDEO_DECODE_READ_BIT_KHR
  - VK_ACCESS_2_VIDEO_DECODE_WRITE_BIT_KHR
- Extending VkBufferUsageFlagBits:
  - VK_BUFFER_USAGE_VIDEO_DECODE_DST_BIT_KHR
  - VK_BUFFER_USAGE_VIDEO_DECODE_SRC_BIT_KHR
- Extending VkFormatFeatureFlagBits:
  - VK_FORMAT_FEATURE_VIDEO_DECODE_DPB_BIT_KHR
  - VK_FORMAT_FEATURE_VIDEO_DECODE_OUTPUT_BIT_KHR
- Extending VkImageLayout:
  - VK_IMAGE_LAYOUT_VIDEO_DECODE_DPB_KHR
  - VK_IMAGE_LAYOUT_VIDEO_DECODE_DST_KHR
  - VK_IMAGE_LAYOUT_VIDEO_DECODE_SRC_KHR
- Extending VkImageUsageFlagBits:
  - VK_IMAGE_USAGE_VIDEO_DECODE_DPB_BIT_KHR
  - VK_IMAGE_USAGE_VIDEO_DECODE_DST_BIT_KHR
  - VK_IMAGE_USAGE_VIDEO_DECODE_SRC_BIT_KHR
- Extending VkPipelineStageFlagBits2KHR:
  - VK_PIPELINE_STAGE_2_VIDEO_DECODE_BIT_KHR
- Extending VkQueueFlagBits:
  - VK_QUEUE_VIDEO_DECODE_BIT_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_VIDEO_DECODE_INFO_KHR
**Version History**

- Revision 1, 2018-6-11 (Peter Fang)
  - Initial draft
- Revision 1.5, Nov 09 2018 (Tony Zlatinski)
  - API Updates
- Revision 1.6, Jan 08 2020 (Tony Zlatinski)
  - API unify with the video_encode_queue spec
- Revision 1.7, March 29 2021 (Tony Zlatinski)
  - Spec and API updates.

**VK_KHR_video_encode_queue**

**Name String**

VK_KHR_video_encode_queue

**Extension Type**

Device extension

**Registered Extension Number**

300

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_video_queue
- Requires VK_KHR_synchronization2

  * This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

**Contact**

- ahmed.abdelkalek@amd.com

**Other Extension Metadata**

**Last Modified Date**

2021-03-29

**IP Status**

No known IP claims.
Contributors

- Ahmed Abdelkhalek, AMD
- Damien Kessler, NVIDIA
- Daniel Rakos, AMD
- George Hao, AMD
- Jake Beju, AMD
- Peter Fang, AMD
- Piers Daniell, NVIDIA
- Srinath Kumarapuram, NVIDIA
- Thomas J. Meier, NVIDIA
- Tony Zlatinski, NVIDIA
- Yang Liu, AMD

New Commands

- \texttt{vkCmdEncodeVideoKHR}

New Structures

- \texttt{VkVideoEncodeInfoKHR}
- Extending \texttt{VkVideoCodingControlInfoKHR}:
  - \texttt{VkVideoEncodeRateControlInfoKHR}

New Enums

- \texttt{VkVideoEncodeFlagBitsKHR}
- \texttt{VkVideoEncodeRateControlFlagBitsKHR}
- \texttt{VkVideoEncodeRateControlModeFlagBitsKHR}

New Bitmasks

- \texttt{VkVideoEncodeFlagsKHR}
- \texttt{VkVideoEncodeRateControlFlagsKHR}
- \texttt{VkVideoEncodeRateControlModeFlagsKHR}

New Enum Constants

- \texttt{VK_KHR_VIDEO_ENCODE_QUEUE_EXTENSION_NAME}
- \texttt{VK_KHR_VIDEO_ENCODE_QUEUE_SPEC_VERSION}
- Extending \texttt{VkAccessFlagBits2KHR}:
  - \texttt{VK_ACCESS_2_VIDEO_ENCODE_READ_BIT_KHR}
• Extending VkBufferUsageFlagBits:
  ◦ VK_BUFFER_USAGE_VIDEO_ENCODE_DST_BIT_KHR
  ◦ VK_BUFFER_USAGE_VIDEO_ENCODE_SRC_BIT_KHR

• Extending VkFormatFeatureFlagBits:
  ◦ VK_FORMAT_FEATURE_VIDEO_ENCODE_DPB_BIT_KHR
  ◦ VK_FORMAT_FEATURE_VIDEO_ENCODE_INPUT_BIT_KHR

• Extending VkImageLayout:
  ◦ VK_IMAGE_LAYOUT_VIDEO_ENCODE_DPB_KHR
  ◦ VK_IMAGE_LAYOUT_VIDEO_ENCODE_DST_KHR
  ◦ VK_IMAGE_LAYOUT_VIDEO_ENCODE_SRC_KHR

• Extending VkImageUsageFlagBits:
  ◦ VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR
  ◦ VK_IMAGE_USAGE_VIDEO_ENCODE_DST_BIT_KHR
  ◦ VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR

• Extending VkPipelineStageFlagBits2KHR:
  ◦ VK_PIPELINE_STAGE_2_VIDEO_ENCODE_BIT_KHR

• Extending VkQueryType:
  ◦ VK_QUERY_TYPE_VIDEO_ENCODE_BITSTREAM_BUFFER_RANGE_KHR

• Extending VkQueueFlagBits:
  ◦ VK_QUEUE_VIDEO_ENCODE_BIT_KHR

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_VIDEO_ENCODE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_VIDEO_ENCODE_RATE_CONTROL_INFO_KHR

**Version History**

• Revision 1, 2018-07-23 (Ahmed Abdelkhalek)
  ◦ Initial draft

• Revision 1.1, 10/29/2019 (Tony Zlatinski)
  ◦ Updated the reserved spec tokens and renamed VkVideoEncoderKHR to VkVideoSessionKHR

• Revision 1.6, Jan 08 2020 (Tony Zlatinski)
  ◦ API unify with the video_decode_queue spec

• Revision 2, March 29 2021 (Tony Zlatinski)
  ◦ Spec and API updates.
VK_KHR_video_queue

Name String
VK_KHR_video_queue

Extension Type
Device extension

Registered Extension Number
24

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2
• Requires VK_KHR_sampler_ycbcr_conversion
• This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

Contact
• Tony Zlatinski tzlatinski

Other Extension Metadata

Last Modified Date
2021-03-29

IP Status
No known IP claims.

Contributors
• Ahmed Abdelkhalek, AMD
• George Hao, AMD
• Jake Beju, AMD
• Piers Daniell, NVIDIA
• Srinath Kumarapuram, NVIDIA
• Tobias Hector, AMD
• Tony Zlatinski, NVIDIA

New Object Types
• VkVideoSessionKHR
New Commands

- `vkBindVideoSessionMemoryKHR`
- `vkCmdBeginVideoCodingKHR`
- `vkCmdControlVideoCodingKHR`
- `vkCmdEndVideoCodingKHR`
- `vkCreateVideoSessionKHR`
- `vkCreateVideoSessionParametersKHR`
- `vkDestroyVideoSessionKHR`
- `vkDestroyVideoSessionParametersKHR`
- `vkGetPhysicalDeviceVideoCapabilitiesKHR`
- `vkGetPhysicalDeviceVideoFormatPropertiesKHR`
- `vkGetVideoSessionMemoryRequirementsKHR`
- `vkUpdateVideoSessionParametersKHR`

New Structures

- `VkPhysicalDeviceVideoFormatInfoKHR`
- `VkVideoBeginCodingInfoKHR`
- `VkVideoBindMemoryKHR`
- `VkVideoCapabilitiesKHR`
- `VkVideoCodingControlInfoKHR`
- `VkVideoEndCodingInfoKHR`
- `VkVideoFormatPropertiesKHR`
- `VkVideoGetMemoryPropertiesKHR`
- `VkVideoPictureResourceKHR`
- `VkVideoReferenceSlotKHR`
- `VkVideoSessionCreateInfoKHR`
- `VkVideoSessionParametersCreateInfoKHR`
- `VkVideoSessionParametersUpdateInfoKHR`

Extending `VkFormatProperties2`, `VkImageCreateInfo`, `VkImageViewCreateInfo`, `VkBufferCreateInfo`:

- `VkVideoProfilesKHR`

Extending `VkQueryPoolCreateInfo`, `VkFormatProperties2`, `VkImageCreateInfo`, `VkImageViewCreateInfo`, `VkBufferCreateInfo`:

- `VkVideoProfileKHR`
• Extending \texttt{VkQueueFamilyProperties2}:
  ◦ \texttt{VkVideoQueueFamilyProperties2KHR}

\section*{New Enums}

• \texttt{VkQueryResultStatusKHR}
• \texttt{VkVideoCapabilitiesFlagBitsKHR}
• \texttt{VkVideoChromaSubsamplingFlagBitsKHR}
• \texttt{VkVideoCodecOperationFlagBitsKHR}
• \texttt{VkVideoCodingControlFlagBitsKHR}
• \texttt{VkVideoCodingQualityPresetFlagBitsKHR}
• \texttt{VkVideoComponentBitDepthFlagBitsKHR}
• \texttt{VkVideoSessionCreateFlagBitsKHR}

\section*{New Bitmasks}

• \texttt{VkVideoBeginCodingFlagsKHR}
• \texttt{VkVideoCapabilitiesFlagsKHR}
• \texttt{VkVideoChromaSubsamplingFlagsKHR}
• \texttt{VkVideoCodecOperationFlagsKHR}
• \texttt{VkVideoCodingControlFlagsKHR}
• \texttt{VkVideoCodingQualityPresetFlagsKHR}
• \texttt{VkVideoComponentBitDepthFlagsKHR}
• \texttt{VkVideoEndCodingFlagsKHR}
• \texttt{VkVideoSessionCreateFlagsKHR}

\section*{New Enum Constants}

• \texttt{VK_KHR_VIDEO_QUEUE_EXTENSION_NAME}
• \texttt{VK_KHR_VIDEO_QUEUE_SPEC_VERSION}

• Extending \texttt{VkObjectType}:
  ◦ \texttt{VK_OBJECT_TYPE_VIDEO_SESSION_KHR}
  ◦ \texttt{VK_OBJECT_TYPE_VIDEO_SESSION_PARAMETERS_KHR}

• Extending \texttt{VkQueryResultFlagBits}:
  ◦ \texttt{VK_QUERY_RESULT_WITH_STATUS_BIT_KHR}

• Extending \texttt{VkQueryType}:
  ◦ \texttt{VK_QUERY_TYPE_RESULT_STATUS_ONLY_KHR}

• Extending \texttt{VkStructureType}:
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VIDEO_FORMAT_INFO_KHR
VK_STRUCTURE_TYPE_VIDEO_BEGIN_CODING_INFO_KHR
VK_STRUCTURE_TYPE_VIDEO_BIND_MEMORY_KHR
VK_STRUCTURE_TYPE_VIDEO_CAPABILITIES_KHR
VK_STRUCTURE_TYPE_VIDEO_CODING_CONTROL_INFO_KHR
VK_STRUCTURE_TYPE_VIDEO_END_CODING_INFO_KHR
VK_STRUCTURE_TYPE_VIDEO_FORMAT_PROPERTIES_KHR
VK_STRUCTURE_TYPE_VIDEO_GET_MEMORY_PROPERTIES_KHR
VK_STRUCTURE_TYPE_VIDEO_PICTURE_RESOURCE_KHR
VK_STRUCTURE_TYPE_VIDEO_PROFILES_KHR
VK_STRUCTURE_TYPE_VIDEO_PROFILE_KHR
VK_STRUCTURE_TYPE_VIDEO_QUEUE_FAMILY_PROPERTIES_2_KHR
VK_STRUCTURE_TYPE_VIDEO_REFERENCE_SLOT_KHR
VK_STRUCTURE_TYPE_VIDEO_SESSION_CREATE_INFO_KHR
VK_STRUCTURE_TYPE_VIDEO_SESSION_PARAMETERS_CREATE_INFO_KHR
VK_STRUCTURE_TYPE_VIDEO_SESSION_PARAMETERS_UPDATE_INFO_KHR

Version History

- Revision 0.1, 2019-11-21 (Tony Zlatinski)
  - Initial draft
- Revision 0.2, 2019-11-27 (Tony Zlatinski)
  - Make vulkan video core common between decode and encode
- Revision 1, March 29 2021 (Tony Zlatinski)
  - Spec and API updates.

VK_EXT_video_decode_h264

Name String

VK_EXT_video_decode_h264

Extension Type

Device extension

Registered Extension Number

41

Revision

1
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_video_decode_queue`
- This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

Contact

- peter.fang@amd.com

Other Extension Metadata

Last Modified Date
2021-03-29

IP Status
No known IP claims.

Contributors

- Chunbo Chen, Intel
- HoHin Lau, AMD
- Jake Beju, AMD
- Peter Fang, AMD
- Ping Liu, Intel
- Srinath Kumarapuram, NVIDIA
- Tony Zlatinski, NVIDIA

New Structures

- Extending `VkVideoCapabilitiesKHR`:
  - `VkVideoDecodeH264CapabilitiesEXT`
- Extending `VkVideoDecodeH264PictureInfoEXT`:
  - `VkVideoDecodeH264MvcEXT`
- Extending `VkVideoDecodeInfoKHR`:
  - `VkVideoDecodeH264PictureInfoEXT`
- Extending `VkVideoProfileKHR`:
  - `VkVideoDecodeH264ProfileEXT`
- Extending `VkVideoReferenceSlotKHR`:
  - `VkVideoDecodeH264DpbSlotInfoEXT`
- Extending `VkVideoSessionCreateInfoKHR`:
  - `VkVideoDecodeH264SessionCreateInfoEXT`
• Extending VkVideoSessionParametersCreateInfoKHR:
  ◦ VkVideoDecodeH264SessionParametersCreateInfoEXT

• Extending VkVideoSessionParametersUpdateInfoKHR:
  ◦ VkVideoDecodeH264SessionParametersAddInfoEXT

New Enums

• VkVideoDecodeH264FieldLayoutFlagBitsEXT

New Bitmasks

• VkVideoDecodeH264CreateFlagsEXT
• VkVideoDecodeH264FieldLayoutFlagsEXT

New Enum Constants

• VK_EXT_VIDEO_DECODE_H264_EXTENSION_NAME
• VK_EXT_VIDEO_DECODE_H264_SPEC_VERSION

• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_CAPABILITIES_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_DPB_SLOT_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_MVC_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_PICTURE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_PROFILE_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_CREATE_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_PARAMETERS_ADD_INFO_EXT
  ◦ VK_STRUCTURE_TYPE_VIDEO_DECODE_H264_SESSION_PARAMETERS_CREATE_INFO_EXT

• Extending VkVideoCodecOperationFlagBitsKHR:
  ◦ VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_EXT

Version History

• Revision 1, 2018-6-11 (Peter Fang)
  ◦ Initial draft

• Revision 1.6, March 29 2021 (Tony Zlatinski)
  ◦ Spec and API Updates

**VK_EXT_video_decode_h265**

Name String

VK_EXT_video_decode_h265
Extension Type
Device extension

Registered Extension Number
188

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_video_decode_queue
• This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

Contact
• peter.fang@amd.com

Other Extension Metadata

Last Modified Date
2021-03-29

IP Status
No known IP claims.

Contributors
• HoHin Lau, AMD
• Jake Beju, AMD
• Peter Fang, AMD
• Ping Liu, Intel
• Srinath Kumarapuram, NVIDIA
• Tony Zlatinski, NVIDIA

New Structures
• Extending VkVideoCapabilitiesKHR:
  ◦ VkVideoDecodeH265CapabilitiesEXT
• Extending VkVideoDecodeInfoKHR:
  ◦ VkVideoDecodeH265PictureInfoEXT
• Extending VkVideoProfileKHR:
  ◦ VkVideoDecodeH265ProfileEXT
• Extending VkVideoReferenceSlotKHR:
• `VkVideoDecodeH265DpbSlotInfoEXT`

• Extending `VkVideoSessionCreateInfoKHR`:
  • `VkVideoDecodeH265SessionCreateInfoEXT`

• Extending `VkVideoSessionParametersCreateInfoKHR`:
  • `VkVideoDecodeH265SessionParametersCreateInfoEXT`

• Extending `VkVideoSessionParametersUpdateInfoKHR`:
  • `VkVideoDecodeH265SessionParametersAddInfoEXT`

**New Bitmasks**

• `VkVideoDecodeH265CreateFlagsEXT`

**New Enum Constants**

• `VK_EXT_VIDEO_DECODE_H265_EXTENSION_NAME`
• `VK_EXT_VIDEO_DECODE_H265_SPEC_VERSION`

• Extending `VkStructureType`:
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_CAPABILITIES_EXT`
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_DPB_SLOT_INFO_EXT`
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_PICTURE_INFO_EXT`
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_PROFILE_EXT`
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_CREATE_INFO_EXT`
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_PARAMETERS_ADD_INFO_EXT`
  • `VK_STRUCTURE_TYPE_VIDEO_DECODE_H265_SESSION_PARAMETERS_CREATE_INFO_EXT`

• Extending `VkVideoCodecOperationFlagBitsKHR`:
  • `VK_VIDEO_CODEC_OPERATION_DECODE_H265_BIT_EXT`

**Version History**

• Revision 1, 2018-6-11 (Peter Fang)
  • Initial draft

• Revision 1.6, March 29 2021 (Tony Zlatinski)
  • Spec and API updates.

**VK_EXT_video_encode_h264**

**Name String**

`VK_EXT_video_encode_h264`
Extension Type
  Device extension

Registered Extension Number
  39

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires VK_KHR_video_encode_queue
  • This is a provisional extension and must be used with caution. See the description of provisional header files for enablement and stability details.

Contact
  • Ahmed Abdelkhalek @aabd Kelh

Other Extension Metadata

Last Modified Date
  2021-03-29

IP Status
  No known IP claims.

Contributors
  • Ahmed Abdelkhalek, AMD
  • Daniel Rakos, AMD
  • George Hao, AMD
  • Jake Beju, AMD
  • Peter Fang, AMD
  • Ping Liu, Intel
  • Srinath Kumarapuram, NVIDIA
  • Tony Zlatinski, NVIDIA
  • Yang Liu, AMD

New Structures
  • VkVideoEncodeH264DpbSlotInfoEXT
  • VkVideoEncodeH264NaluSliceEXT
  • Extending VkVideoCapabilitiesKHR:
    ◦ VkVideoEncodeH264CapabilitiesEXT
• Extending VkVideoEncodeInfoKHR:
  ◦ VkVideoEncodeH264EmitPictureParametersEXT
  ◦ VkVideoEncodeH264VclFrameInfoEXT

• Extending VkVideoProfileKHR:
  ◦ VkVideoEncodeH264ProfileEXT

• Extending VkVideoSessionCreateInfoKHR:
  ◦ VkVideoEncodeH264SessionCreateInfoEXT

• Extending VkVideoSessionParametersCreateInfoKHR:
  ◦ VkVideoEncodeH264SessionParametersCreateInfoEXT

• Extending VkVideoSessionParametersUpdateInfoKHR:
  ◦ VkVideoEncodeH264SessionParametersAddInfoEXT

New Enums

• VkVideoEncodeH264CapabilitiesFlagBitsEXT
• VkVideoEncodeH264CreateFlagBitsEXT
• VkVideoEncodeH264InputModeFlagBitsEXT
• VkVideoEncodeH264OutputModeFlagBitsEXT

New Bitmasks

• VkVideoEncodeH264CapabilitiesFlagsEXT
• VkVideoEncodeH264CreateFlagsEXT
• VkVideoEncodeH264InputModeFlagsEXT
• VkVideoEncodeH264OutputModeFlagsEXT

New Enum Constants

• VK_EXT_VIDEO_ENCODE_H264_EXTENSION_NAME
• VK_EXT_VIDEO_ENCODE_H264_SPEC_VERSION

Extending VkStructureType:

• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_CAPABILITIES_EXT
• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_DPB_SLOT_INFO_EXT
• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_EMIT_PICTURE_PARAMETERS_EXT
• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_NALU_SLICE_EXT
• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_PROFILE_EXT
• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_CREATE_INFO_EXT
• VK_STRUCTURE_TYPE_VIDEO_ENCODE_H264_SESSION_PARAMETERS_ADD_INFO_EXT
Extending `VkVideoCodecOperationFlagBitsKHR`:
- `VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_EXT`

**Version History**

- Revision 0, 2018-7-23 (Ahmed Abdelkhalek)
  - Initial draft
- Revision 0.5, 2020-02-13 (Tony Zlatinski)
  - General Spec cleanup
  - Added DPB structures
  - Change the VCL frame encode structure
  - Added a common Non-VCL Picture Parameters structure
- Revision 1, 2021-03-29 (Tony Zlatinski)
  - Spec and API updates

**List of Deprecated Extensions**

- `VK_KHR_16bit_storage`
- `VK_KHR_bind_memory2`
- `VK_KHR_dedicated_allocation`
- `VK_KHR_descriptor_update_template`
- `VK_KHR_device_group`
- `VK_KHR_device_group_creation`
- `VK_KHR_external_fence`
- `VK_KHR_external_fence_capabilities`
- `VK_KHR_external_memory`
- `VK_KHR_external_memory_capabilities`
- `VK_KHR_external_semaphore`
- `VK_KHR_external_semaphore_capabilities`
- `VK_KHR_get_memory_requirements2`
- `VK_KHR_get_physical_device_properties2`
- `VK_KHR_maintenance1`
- `VK_KHR_maintenance2`
- `VK_KHR_maintenance3`
- `VK_KHR_multiview`
• VK_KHR_relaxed_block_layout
• VK_KHR_sampler_ycbcr_conversion
• VK_KHR_shader_draw_parameters
• VK_KHR_storage_buffer_storage_class
• VK_KHR_variable_pointers
• VK_EXT_buffer_device_address
• VK_EXT_debug_marker
• VK_EXT_debug_report
• VK_EXT_shader_subgroup_vote
• VK_EXT_validation_flags
• VK_AMD_draw_indirect_count
• VK_AMD_gpu_shader_half_float
• VK_AMD_gpu_shader_int16
• VK_AMD_negative_viewport_height
• VK_MVK_ios_surface
• VK_MVK_macos_surface
• VK_NV_dedicated_allocation
• VK_NV_external_memory
• VK_NV_external_memory_capabilities
• VK_NV_external_memory_win32
• VK_NV_gls1_shader
• VK_NV_win32_keyed_mutex
VK_KHR_16bit_storage

Name String
VK_KHR_16bit_storage

Extension Type
Device extension

Registered Extension Number
84

Revision
1

Extension and Version Dependencies
- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2
- Requires VK_KHR_storage_buffer_storage_class

Deprecation state
- Promoted to Vulkan 1.1

Contact
- Jan-Harald Fredriksen janharaldfredriksen-arm

Other Extension Metadata

Last Modified Date
2017-09-05

IP Status
No known IP claims.

Interactions and External Dependencies
- Promoted to Vulkan 1.1 Core
- This extension requires SPV_KHR_16bit_storage
- This extension provides API support for GL_EXT_shader_16bit_storage

Contributors
- Alexander Galazin, ARM
- Jan-Harald Fredriksen, ARM
- Joerg Wagner, ARM
- Neil Henning, Codeplay
- Jeff Bolz, Nvidia
- Daniel Koch, Nvidia
Description

The VK_KHR_16bit_storage extension allows use of 16-bit types in shader input and output interfaces, and push constant blocks. This extension introduces several new optional features which map to SPIR-V capabilities and allow access to 16-bit data in Block-decorated objects in the Uniform and the StorageBuffer storage classes, and objects in the PushConstant storage class. This extension allows 16-bit variables to be declared and used as user-defined shader inputs and outputs but does not change location assignment and component assignment rules.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. However, if Vulkan 1.1 is supported and this extension is not, the storageBuffer16BitAccess capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDevice16BitStorageFeaturesKHR

New Enum Constants

- VK_KHR_16BIT_STORAGE_EXTENSION_NAME
- VK_KHR_16BIT_STORAGE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_16BIT_STORAGE_FEATURES_KHR

New SPIR-V Capabilities

- StorageBuffer16BitAccess
- UniformAndStorageBuffer16BitAccess
- StoragePushConstant16
- StorageInputOutput16

Version History

- Revision 1, 2017-03-23 (Alexander Galazin)
  - Initial draft
VK_KHR_bind_memory2

Name String
VK_KHR_bind_memory2

Extension Type
Device extension

Registered Extension Number
158

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Deprecation state
• Promoted to Vulkan 1.1

Contact
• Tobias Hector tobiski

Other Extension Metadata

Last Modified Date
2017-09-05

IP Status
No known IP claims.

Interactions and External Dependencies
• Promoted to Vulkan 1.1 Core

Contributors
• Jeff Bolz, NVIDIA
• Tobias Hector, Imagination Technologies

Description
This extension provides versions of vkBindBufferMemory and vkBindImageMemory that allow multiple bindings to be performed at once, and are extensible.

This extension also introduces VK_IMAGE_CREATE_ALIAS_BIT_KHR, which allows “identical” images that alias the same memory to interpret the contents consistently, even across image layout changes.
Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- `vkBindBufferMemory2KHR`
- `vkBindImageMemory2KHR`

New Structures

- `VkBindBufferMemoryInfoKHR`
- `VkBindImageMemoryInfoKHR`

New Enum Constants

- `VK_KHR_BIND_MEMORY_2_EXTENSION_NAME`
- `VK_KHR_BIND_MEMORY_2_SPEC_VERSION`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_ALIAS_BIT_KHR`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_INFO_KHR`
  - `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_INFO_KHR`

Version History

- Revision 1, 2017-05-19 (Tobias Hector)
  - Pulled bind memory functions into their own extension

VK_KHR_dedicated_allocation

Name String

`VK_KHR_dedicated_allocation`

Extension Type

Device extension

Registered Extension Number

128

Revision

3
Extension and Version Dependencies
  - Requires Vulkan 1.0
  - Requires VK_KHR_get_memory_requirements2

Deprecation state
  - Promoted to Vulkan 1.1

Contact
  - James Jones

Other Extension Metadata

Last Modified Date
  2017-09-05

IP Status
  No known IP claims.

Interactions and External Dependencies
  - Promoted to Vulkan 1.1 Core

Contributors
  - Jeff Bolz, NVIDIA
  - Jason Ekstrand, Intel

Description

This extension enables resources to be bound to a dedicated allocation, rather than suballocated. For any particular resource, applications can query whether a dedicated allocation is recommended, in which case using a dedicated allocation may improve the performance of access to that resource. Normal device memory allocations must support multiple resources per allocation, memory aliasing and sparse binding, which could interfere with some optimizations. Applications should query the implementation for when a dedicated allocation may be beneficial by adding a VkMemoryDedicatedRequirementsKHR structure to the pNext chain of the VkMemoryRequirements2 structure passed as the pMemoryRequirements parameter of a call to vkGetBufferMemoryRequirements2 or vkGetImageMemoryRequirements2. Certain external handle types and external images or buffers may also depend on dedicated allocations on implementations that associate image or buffer metadata with OS-level memory objects.

This extension adds a two small structures to memory requirements querying and memory allocation: a new structure that flags whether an image/buffer should have a dedicated allocation, and a structure indicating the image or buffer that an allocation will be bound to.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.
New Structures

- Extending *VkMemoryAllocateInfo*:
  - *VkMemoryDedicatedAllocateInfoKHR*

- Extending *VkMemoryRequirements2*:
  - *VkMemoryDedicatedRequirementsKHR*

New Enum Constants

- *VK_KHR_DEDICATED_ALLOCATION_EXTENSION_NAME*
- *VK_KHR_DEDICATED_ALLOCATION_SPEC_VERSION*

- Extending *VkStructureType*:
  - *VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR*
  - *VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS_KHR*

Examples

```c
// Create an image with a dedicated allocation based on the
// implementation's preference

VkImageCreateInfo imageCreateInfo =
{
  // Image creation parameters
};

VkImage image;
VkResult result = vkCreateImage(
  device,
  &imageCreateInfo,
  NULL,          // pAllocator
  &image);

VkMemoryDedicatedRequirementsKHR dedicatedRequirements =
{
  VK_STRUCTURE_TYPE_MEMORY_DEDICATED_REQUIREMENTS_KHR,
  NULL,          // pNext
};

VkMemoryRequirements2 memoryRequirements =
{
  VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2,
  &dedicatedRequirements,  // pNext
};

const VkImageMemoryRequirementsInfo2 imageRequirementsInfo =
{
  VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2,
};
```
vkGetImageMemoryRequirements2(
    device,
    &imageRequirementsInfo,
    &memoryRequirements);

if (dedicatedRequirements.prefersDedicatedAllocation) {
    // Allocate memory with VkMemoryDedicatedAllocateInfoKHR::image
    // pointing to the image we are allocating the memory for

    VkMemoryDedicatedAllocateInfoKHR dedicatedInfo =
    {
        VK_STRUCTURE_TYPE_MEMORY_DEDICATED_ALLOCATE_INFO_KHR,  // sType
        NULL,  // pNext
        image,  // image
        VK_NULL_HANDLE,  // buffer
    };

    VkMemoryAllocateInfo memoryAllocateInfo =
    {
        VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,  // sType
        &dedicatedInfo,  // pNext
        memoryRequirements.size,  // allocationSize
        FindMemoryTypeIndex(memoryRequirements.memoryTypeBits),  // memoryTypeIndex
    };

    VkDeviceMemory memory;
    vkAllocateMemory(
        device,
        &memoryAllocateInfo,
        NULL,  // allocator
        &memory);

    // Bind the image to the memory

    vkBindImageMemory(
        device,
        image,
        memory,
        0);
} else {
    // Take the normal memory sub-allocation path
}

Version History

- Revision 1, 2017-02-27 (James Jones)
• Copy content from VK_NV_dedicated_allocation
• Add some references to external object interactions to the overview.

• Revision 2, 2017-03-27 (Jason Ekstrand)
  • Rework the extension to be query-based

• Revision 3, 2017-07-31 (Jason Ekstrand)
  • Clarify that memory objects allocated with VkMemoryDedicatedAllocateInfoKHR can only have the specified resource bound and no others.

**VK_KHR_descriptor_update_template**

**Name String**

VK_KHR_descriptor_update_template

**Extension Type**

Device extension

**Registered Extension Number**

86

**Revision**

1

**Extension and Version Dependencies**

• Requires Vulkan 1.0

**Deprecation state**

• Promoted to Vulkan 1.1

**Contact**

• Markus Tavenrath mtavenrath

**Other Extension Metadata**

**Last Modified Date**

2017-09-05

**IP Status**

No known IP claims.

**Interactions and External Dependencies**

• Interacts with VK_KHR_push_descriptor
  • Promoted to Vulkan 1.1 Core

**Contributors**

• Jeff Bolz, NVIDIA
Description

Applications may wish to update a fixed set of descriptors in a large number of descriptor sets very frequently, i.e. during initialization phase or if it is required to rebuild descriptor sets for each frame. For those cases it is also not unlikely that all information required to update a single descriptor set is stored in a single struct. This extension provides a way to update a fixed set of descriptors in a single `VkDescriptorSet` with a pointer to a user defined data structure describing the new descriptors.

**Promotion to Vulkan 1.1**

`vkCmdPushDescriptorSetWithTemplateKHR` is included as an interaction with `VK_KHR_push_descriptor`. If Vulkan 1.1 and `VK_KHR_push_descriptor` are supported, this is included by `VK_KHR_push_descriptor`.

The base functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

**New Object Types**

- `VkDescriptorUpdateTemplateKHR`

**New Commands**

- `vkCreateDescriptorUpdateTemplateKHR`
- `vkDestroyDescriptorUpdateTemplateKHR`
- `vkUpdateDescriptorSetWithTemplateKHR`

If `VK_KHR_push_descriptor` is supported:

- `vkCmdPushDescriptorSetWithTemplateKHR`

**New Structures**

- `VkDescriptorUpdateTemplateCreateInfoKHR`
- `VkDescriptorUpdateTemplateEntryKHR`

**New Enums**

- `VkDescriptorUpdateTemplateTypeKHR`

**New Bitmasks**

- `VkDescriptorUpdateTemplateCreateFlagsKHR`
New Enum Constants

- VK_KHR_DESCRIPTOR_UPDATE_TEMPLATE_EXTENSION_NAME
- VK_KHR_DESCRIPTOR_UPDATE_TEMPLATE_SPEC_VERSION

Extending VkDescriptorUpdateTemplateType:
  - VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_DESCRIPTOR_SET_KHR

Extending VkObjectType:
  - VK_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR

Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_CREATE_INFO_KHR

If VK_EXT_debug_report is supported:

- Extending VkDebugReportObjectTypeEXT:
  - VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR_EXT

If VK_KHR_push_descriptor is supported:

- Extending VkDescriptorUpdateTemplateType:
  - VK_DESCRIPTOR_UPDATE_TEMPLATE_TYPE_PUSH_DESCRIPTORS_KHR

Version History

- Revision 1, 2016-01-11 (Markus Tavenrath)
  - Initial draft

VK_KHR_device_group

Name String
  VK_KHR_device_group

Extension Type
  Device extension

Registered Extension Number
  61

Revision
  4

Extension and Version Dependencies
  - Requires Vulkan 1.0
  - Requires VK_KHR_device_group_creation
Deprecation state

- Promoted to Vulkan 1.1

Contact

- Jeff Bolz @jeffbolznv

Other Extension Metadata

Last Modified Date

2017-10-10

IP Status

No known IP claims.

Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

Contributors

- Jeff Bolz, NVIDIA
  - Tobias Hector, Imagination Technologies

Description

This extension provides functionality to use a logical device that consists of multiple physical devices, as created with the VK_KHR_device_group_creation extension. A device group can allocate memory across the subdevices, bind memory from one subdevice to a resource on another subdevice, record command buffers where some work executes on an arbitrary subset of the subdevices, and potentially present a swapchain image from one or more subdevices.

Promotion to Vulkan 1.1

The following enums, types and commands are included as interactions with VK_KHR_swapchain:

- VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR
- VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR
- VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR
- VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR
- VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR
- VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR
- VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR
- VkDeviceGroupPresentFlagsKHR
- VkDeviceGroupPresentCapabilitiesKHR
- VkImageSwapchainCreateInfoKHR
- VkBindImageMemorySwapchainInfoKHR
• VkAcquireNextImageInfoKHR
• VkDeviceGroupPresentInfoKHR
• VkDeviceGroupSwapchainCreateInfoKHR
• vkGetDeviceGroupPresentCapabilitiesKHR
• vkGetDeviceGroupSurfacePresentModesKHR
• vkGetPhysicalDevicePresentRectanglesKHR
• vkAcquireNextImage2KHR

If Vulkan 1.1 and VK_KHR_swapchain are supported, these are included by VK_KHR_swapchain.

The base functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

**New Commands**

• vkCmdDispatchBaseKHR
• vkCmdSetDeviceMaskKHR
• vkGetDeviceGroupPeerMemoryFeaturesKHR

If VK_KHR_surface is supported:

• vkGetDeviceGroupPresentCapabilitiesKHR
• vkGetDeviceGroupSurfacePresentModesKHR
• vkGetPhysicalDevicePresentRectanglesKHR

If VK_KHR_swapchain is supported:

• vkAcquireNextImage2KHR

**New Structures**

• Extending VkBindSparseInfo:
  ◦ VkDeviceGroupBindSparseInfoKHR

• Extending VkCommandBufferBeginInfo:
  ◦ VkDeviceGroupCommandBufferBeginInfoKHR

• Extending VkMemoryAllocateInfo:
  ◦ VkMemoryAllocateFlagsInfoKHR

• Extending VkRenderPassBeginInfo:
  ◦ VkDeviceGroupRenderPassBeginInfoKHR

• Extending VkSubmitInfo:
  ◦ VkDeviceGroupSubmitInfoKHR
If VK_KHR_bind_memory2 is supported:

- Extending VkBindBufferMemoryInfo:
  - VkBindBufferMemoryDeviceGroupInfoKHR
- Extending VkBindImageMemoryInfo:
  - VkBindImageMemoryDeviceGroupInfoKHR

If VK_KHR_surface is supported:

- VkDeviceGroupPresentCapabilitiesKHR

If VK_KHR_swapchain is supported:

- VkAcquireNextImageInfoKHR
- Extending VkBindImageMemoryInfo:
  - VkBindImageMemorySwapchainInfoKHR
- Extending VkImageCreateInfo:
  - VkImageSwapchainCreateInfoKHR
- Extending VkPresentInfoKHR:
  - VkDeviceGroupPresentInfoKHR
- Extending VkSwapchainCreateInfoKHR:
  - VkDeviceGroupSwapchainCreateInfoKHR

New Enums

- VkMemoryAllocateFlagBitsKHR
- VkPeerMemoryFeatureFlagBitsKHR

If VK_KHR_surface is supported:

- VkDeviceGroupPresentModeFlagBitsKHR

New Bitmasks

- VkMemoryAllocateFlagsKHR
- VkPeerMemoryFeatureFlagsKHR

If VK_KHR_surface is supported:

- VkDeviceGroupPresentModeFlagsKHR

New Enum Constants

- VK_KHR_DEVICE_GROUP_EXTENSION_NAME
- VK_KHR_DEVICE_GROUP_SPEC_VERSION
• Extending `VkDependencyFlagBits`:
  ◦ `VK_DEPENDENCY_DEVICE_GROUP_BIT_KHR`

• Extending `VkMemoryAllocateFlagBits`:
  ◦ `VK_MEMORY_ALLOCATE_DEVICE_MASK_BIT_KHR`

• Extending `VkPeerMemoryFeatureFlagBits`:
  ◦ `VK_PEER_MEMORY_FEATURE_COPY_DST_BIT_KHR`
  ◦ `VK_PEER_MEMORY_FEATURE_COPY_SRC_BIT_KHR`
  ◦ `VK_PEER_MEMORY_FEATURE_GENERIC_DST_BIT_KHR`
  ◦ `VK_PEER_MEMORY_FEATURE_GENERIC_SRC_BIT_KHR`

• Extending `VkPipelineCreateFlagBits`:
  ◦ `VK_PIPELINE_CREATE_DISPATCH_BASE_KHR`
  ◦ `VK_PIPELINE_CREATE_VIEW_INDEX_FROM_DEVICE_INDEX_BIT_KHR`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_BIND_SPARSE_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_COMMAND_BUFFER_BEGIN_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_RENDER_PASS_BEGIN_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_SUBMIT_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_FLAGS_INFO_KHR`

If `VK_KHR_bind_memory2` is supported:

• Extending `VkImageCreateFlagBits`:
  ◦ `VK_IMAGE_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_BIND_BUFFER_MEMORY_DEVICE_GROUP_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_DEVICE_GROUP_INFO_KHR`

If `VK_KHR_surface` is supported:

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_CAPABILITIES_KHR`

If `VK_KHR_swapchain` is supported:

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_ACQUIRE_NEXT_IMAGE_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_BIND_IMAGE_MEMORY_SWAPCHAIN_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_PRESENT_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_DEVICE_GROUP_SWAPCHAIN_CREATE_INFO_KHR`
**VK_STRUCTURE_TYPE_IMAGE_SWAPCHAIN_CREATE_INFO_KHR**

- Extending `VkSwapchainCreateFlagBitsKHR`:
  - `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`

### New Built-in Variables

- `DeviceIndex`

### New SPIR-V Capabilities

- `DeviceGroup`

### Version History

- Revision 1, 2016-10-19 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2017-05-19 (Tobias Hector)
  - Removed extended memory bind functions to `VK_KHR_bind_memory2`, added dependency on that extension, and device-group-specific structs for those functions.
- Revision 3, 2017-10-06 (Ian Elliott)
  - Corrected Vulkan 1.1 interactions with the WSI extensions. All Vulkan 1.1 WSI interactions are with the `VK_KHR_swapchain` extension.
- Revision 4, 2017-10-10 (Jeff Bolz)
  - Rename “SFR” bits and structure members to use the phrase “split instance bind regions”.

---

**VK_KHR_device_group_creation**

**Name String**

`VK_KHR_device_group_creation`

**Extension Type**

Instance extension

**Registered Extension Number**

71

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Deprecation state**

- **Promoted** to Vulkan 1.1
Description

This extension provides instance-level commands to enumerate groups of physical devices, and to create a logical device from a subset of one of those groups. Such a logical device can then be used with new features in the `VK_KHR_device_group` extension.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- `vkEnumeratePhysicalDeviceGroupsKHR`

New Structures

- `VkPhysicalDeviceGroupPropertiesKHR`
- Extending `VkDeviceCreateInfo`:
  - `VkDeviceGroupDeviceCreateInfoKHR`

New Enum Constants

- `VK_KHR_DEVICE_GROUP_CREATION_EXTENSION_NAME`
- `VK_KHR_DEVICE_GROUP_CREATION_SPEC_VERSION`
- `VK_MAX_DEVICE_GROUP_SIZE_KHR`
- Extending `VkMemoryHeapFlagBits`:
  - `VK_MEMORY_HEAP_MULTI_INSTANCE_BIT_KHR`
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHR

Examples

```c
VkDeviceCreateInfo devCreateInfo = { VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };  // (not shown) fill out devCreateInfo as usual.
uint32_t deviceGroupCount = 0;
VkPhysicalDeviceGroupPropertiesKHR *props = NULL;

// Query the number of device groups
vkEnumeratePhysicalDeviceGroupsKHR(g_vkInstance, &deviceGroupCount, NULL);

// Allocate and initialize structures to query the device groups
props = (VkPhysicalDeviceGroupPropertiesKHR *)malloc(deviceGroupCount*sizeof(VkPhysicalDeviceGroupPropertiesKHR));
for (i = 0; i < deviceGroupCount; ++i) {
    props[i].sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_GROUP_PROPERTIES_KHR;
    props[i].pNext = NULL;
}
vkEnumeratePhysicalDeviceGroupsKHR(g_vkInstance, &deviceGroupCount, props);

// If the first device group has more than one physical device. create
// a logical device using all of the physical devices.
VkDeviceGroupDeviceCreateInfoKHR deviceGroupInfo = {
    VK_STRUCTURE_TYPE_DEVICE_GROUP_DEVICE_CREATE_INFO_KHR
};
if (props[0].physicalDeviceCount > 1) {
    deviceGroupInfo.physicalDeviceCount = props[0].physicalDeviceCount;
    deviceGroupInfo.pPhysicalDevices = props[0].physicalDevices;
    devCreateInfo.pNext = &deviceGroupInfo;
}
vkCreateDevice(props[0].physicalDevices[0], &devCreateInfo, NULL, &g_vkDevice);
free(props);
```

Version History

• Revision 1, 2016-10-19 (Jeff Bolz)
  ◦ Internal revisions

VK_KHR_external_fence

Name String
  VK_KHR_external_fence
Extension Type
  Device extension

Registered Extension Number
  114

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0
  • Requires `VK_KHR_external_fence_capabilities`

Deprecation state
  • Promoted to Vulkan 1.1

Contact
  • Jesse Hall @critsec

Other Extension Metadata

Last Modified Date
  2017-05-08

IP Status
  No known IP claims.

Interactions and External Dependencies
  • Promoted to Vulkan 1.1 Core

Contributors
  • Jesse Hall, Google
  • James Jones, NVIDIA
  • Jeff Juliano, NVIDIA
  • Cass Everitt, Oculus
  • Contributors to `VK_KHR_external_semaphore`

Description

An application using external memory may wish to synchronize access to that memory using fences. This extension enables an application to create fences from which non-Vulkan handles that reference the underlying synchronization primitive can be exported.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The
original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending `VkFenceCreateInfo`:
  - `VkExportFenceCreateInfoKHR`

New Enums

- `VkFenceImportFlagBitsKHR`

New Bitmasks

- `VkFenceImportFlagsKHR`

New Enum Constants

- `VK_KHR_EXTERNAL_FENCE_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_FENCE_SPEC_VERSION`
- Extending `VkFenceImportFlagBits`:
  - `VK_FENCE_IMPORT_TEMPORARY_BIT_KHR`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_EXPORT_FENCE_CREATE_INFO_KHR`

Issues

This extension borrows concepts, semantics, and language from `VK_KHR_external_semaphore`. That extension's issues apply equally to this extension.

Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial revision

`VK_KHR_external_fence_capabilities`

Name String

- `VK_KHR_external_fence_capabilities`

Extension Type

- Instance extension

Registered Extension Number

113
Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

Deprecation state

- Promoted to Vulkan 1.1

Contact

- Jesse Hall @critsec

Other Extension Metadata

Last Modified Date

2017-05-08

IP Status

No known IP claims.

Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

Contributors

- Jesse Hall, Google
- James Jones, NVIDIA
- Jeff Juliano, NVIDIA
- Cass Everitt, Oculus
- Contributors to `VK_KHR_external_semaphore_capabilities`

Description

An application may wish to reference device fences in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension provides a set of capability queries and handle definitions that allow an application to determine what types of “external” fence handles an implementation supports for a given set of use cases.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- `vkGetPhysicalDeviceExternalFencePropertiesKHR`
New Structures

- VkExternalFencePropertiesKHR
- VkPhysicalDeviceExternalFenceInfoKHR
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceIDPropertiesKHR

New Enums

- VkExternalFenceFeatureFlagBitsKHR
- VkExternalFenceHandleTypeFlagBitsKHR

New Bitmasks

- VkExternalFenceFeatureFlagsKHR
- VkExternalFenceHandleTypeFlagsKHR

New Enum Constants

- VK_KHR_EXTERNAL_FENCE_CAPABILITIES_EXTENSION_NAME
- VK_KHR_EXTERNAL_FENCE_CAPABILITIES_SPEC_VERSION
- VK_LUID_SIZE_KHR
- Extending VkExternalFenceFeatureFlagBits:
  - VK_EXTERNAL_FENCE_FEATURE_EXPORTABLE_BIT_KHR
  - VK_EXTERNAL_FENCE_FEATURE_IMPORTABLE_BIT_KHR
- Extending VkExternalFenceHandleTypeFlagBits:
  - VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_FD_BIT_KHR
  - VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR
  - VK_EXTERNAL_FENCE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR
  - VK_EXTERNAL_FENCE_HANDLE_TYPE_SYNC_FD_BIT_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_EXTERNAL_FENCE_PROPERTIES_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_FENCE_INFO_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHR

Version History

- Revision 1, 2017-05-08 (Jesse Hall)
  - Initial version
VK_KHR_external_memory

Name String
VK_KHR_external_memory

Extension Type
Device extension

Registered Extension Number
73

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_external_memory_capabilities

Deprecation state
• Promoted to Vulkan 1.1

Contact
• James Jones cubanismo

Other Extension Metadata

Last Modified Date
2016-10-20

IP Status
No known IP claims.

Interactions and External Dependencies
• Interacts with VK_KHR_dedicated_allocation.
• Interacts with VK_NV_dedicated_allocation.
• Promoted to Vulkan 1.1 Core

Contributors
• Jason Ekstrand, Intel
• Ian Elliot, Google
• Jesse Hall, Google
• Tobias Hector, Imagination Technologies
• James Jones, NVIDIA
• Jeff Juliano, NVIDIA
• Matthew Netsch, Qualcomm Technologies, Inc.
Description

An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension enables an application to export non-Vulkan handles from Vulkan memory objects such that the underlying resources can be referenced outside the scope of the Vulkan logical device that created them.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending `VkBufferCreateInfo`:
  - `VkExternalMemoryBufferCreateInfoKHR`
- Extending `VkImageCreateInfo`:
  - `VkExternalMemoryImageCreateInfoKHR`
- Extending `VkMemoryAllocateInfo`:
  - `VkExportMemoryAllocateInfoKHR`

New Enum Constants

- `VK_KHR_EXTERNAL_MEMORY_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_MEMORY_SPEC_VERSION`
- `VK_QUEUE_FAMILY_EXTERNAL_KHR`
- Extending `VkResult`:
  - `VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO_KHR`
  - `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_KHR`

Issues

1) How do applications correlate two physical devices across process or Vulkan instance boundaries?
RESOLVED: New device ID fields have been introduced by `VK_KHR_external_memory_capabilities`. These fields, combined with the existing `VkPhysicalDeviceProperties::driverVersion` field can be used to identify compatible devices across processes, drivers, and APIs. `VkPhysicalDeviceProperties::pipelineCacheUUID` is not sufficient for this purpose because despite its description in the specification, it need only identify a unique pipeline cache format in practice. Multiple devices may be able to use the same pipeline cache data, and hence it would be desirable for all of them to have the same pipeline cache UUID. However, only the same concrete physical device can be used when sharing memory, so an actual unique device ID was introduced. Further, the pipeline cache UUID was specific to Vulkan, but correlation with other, non-extensible APIs is required to enable interoperation with those APIs.

2) If memory objects are shared between processes and APIs, is this considered aliasing according to the rules outlined in the Memory Aliasing section?

RESOLVED: Yes. Applications must take care to obey all restrictions imposed on aliased resources when using memory across multiple Vulkan instances or other APIs.

3) Are new image layouts or metadata required to specify image layouts and layout transitions compatible with non-Vulkan APIs, or with other instances of the same Vulkan driver?

RESOLVED: Separate instances of the same Vulkan driver running on the same GPU should have identical internal layout semantics, so applications have the tools they need to ensure views of images are consistent between the two instances. Other APIs will fall into two categories: Those that are Vulkan-compatible, and those that are Vulkan-incompatible. Vulkan-incompatible APIs will require the image to be in the GENERAL layout whenever they are accessing them.

Note this does not attempt to address cross-device transitions, nor transitions to engines on the same device which are not visible within the Vulkan API. Both of these are beyond the scope of this extension.

4) Is a new barrier flag or operation of some type needed to prepare external memory for handoff to another Vulkan instance or API and/or receive it from another instance or API?

RESOLVED: Yes. Some implementations need to perform additional cache management when transitioning memory between address spaces and other APIs, instances, or processes which may operate in a separate address space. Options for defining this transition include:

- A new structure that can be added to the `pNext` list in `VkMemoryBarrier`, `VkBufferMemoryBarrier`, and `VkImageMemoryBarrier`.
- A new bit in `VkAccessFlags` that can be set to indicate an “external” access.
- A new bit in `VkDependencyFlags`.
- A new special queue family that represents an “external” queue.

A new structure has the advantage that the type of external transition can be described in as much detail as necessary. However, there is not currently a known need for anything beyond differentiating between external and internal accesses, so this is likely an over-engineered solution. The access flag bit has the advantage that it can be applied at buffer, image, or global granularity, and semantically it maps pretty well to the operation being described. Additionally, the API already includes `VK_ACCESS_MEMORY_READ_BIT` and `VK_ACCESS_MEMORY_WRITE_BIT` which appear to be intended
for this purpose. However, there is no obvious pipeline stage that would correspond to an external access, and therefore no clear way to use `VK_ACCESS_MEMORY_READ_BIT` or `VK_ACCESS_MEMORY_WRITE_BIT`. `VkDependecyFlags` and `VkPipelineStageFlags` operate at command granularity rather than image or buffer granularity, which would make an entire pipeline barrier an internal→external or external→internal barrier. This may not be a problem in practice, but seems like the wrong scope. Another downside of `VkDependencyFlags` is that it lacks inherent directionality: there are no `src` and `dst` variants of it in the barrier or dependency description semantics, so two bits might need to be added to describe both internal→external and external→internal transitions. Transitioning a resource to a special queue family corresponds well with the operation of transitioning to a separate Vulkan instance, in that both operations ideally include scheduling a barrier on both sides of the transition: Both the releasing and the acquiring queue or process. Using a special queue family requires adding an additional reserved queue family index. Re-using `VK_QUEUE_FAMILY_IGNORED` would have left it unclear how to transition a concurrent usage resource from one process to another, since the semantics would have likely been equivalent to the currently-ignored transition of `VK_QUEUE_FAMILY_IGNORED → VK_QUEUE_FAMILY_IGNORED`. Fortunately, creating a new reserved queue family index is not invasive.

Based on the above analysis, the approach of transitioning to a special “external” queue family was chosen.

5) Do internal driver memory arrangements and/or other internal driver image properties need to be exported and imported when sharing images across processes or APIs.

**RESOLVED:** Some vendors claim this is necessary on their implementations, but it was determined that the security risks of allowing opaque metadata to be passed from applications to the driver were too high. Therefore, implementations which require metadata will need to associate it with the objects represented by the external handles, and rely on the dedicated allocation mechanism to associate the exported and imported memory objects with a single image or buffer.

6) Most prior interoperation and cross-process sharing APIs have been based on image-level sharing. Should Vulkan sharing be based on memory-object sharing or image sharing?

**RESOLVED:** These extensions have assumed memory-level sharing is the correct granularity. Vulkan is a lower-level API than most prior APIs, and as such attempts to closely align with the underlying primitives of the hardware and system-level drivers it abstracts. In general, the resource that holds the backing store for both images and buffers of various types is memory. Images and buffers are merely metadata containing brief descriptions of the layout of bits within that memory.

Because memory object-based sharing is aligned with the overall Vulkan API design, it enables the full range of Vulkan capabilities with external objects. External memory can be used as backing for sparse images, for example, whereas such usage would be awkward at best with a sharing mechanism based on higher-level primitives such as images. Further, aligning the mechanism with the API in this way provides some hope of trivial compatibility with future API enhancements. If new objects backed by memory objects are added to the API, they too can be used across processes with minimal additions to the base external memory APIs.

Earlier APIs implemented interop at a higher level, and this necessitated entirely separate sharing APIs for images and buffers. To co-exist and interoperate with those APIs, the Vulkan external
sharing mechanism must accommodate their model. However, if it can be agreed that memory-based sharing is the more desirable and forward-looking design, legacy interoperation constraints can be considered another reason to favor memory-based sharing: while native and legacy driver primitives that may be used to implement sharing may not be as low-level as the API here suggests, raw memory is still the least common denominator among the types. Image-based sharing can be cleanly derived from a set of base memory-object sharing APIs with minimal effort, whereas image-based sharing does not generalize well to buffer or raw-memory sharing. Therefore, following the general Vulkan design principle of minimalism, it is better to expose interoperability with image-based native and external primitives via the memory sharing API, and place sufficient limits on their usage to ensure they can be used only as backing for equivalent Vulkan images. This provides a consistent API for applications regardless of which platform or external API they are targeting, which makes development of multi-API and multi-platform applications simpler.

7) Should Vulkan define a common external handle type and provide Vulkan functions to facilitate cross-process sharing of such handles rather than relying on native handles to define the external objects?

**RESOLVED:** No. Cross-process sharing of resources is best left to native platforms. There are myriad security and extensibility issues with such a mechanism, and attempting to re-solve all those issues within Vulkan does not align with Vulkan's purpose as a graphics API. If desired, such a mechanism could be built as a layer or helper library on top of the opaque native handle defined in this family of extensions.

8) Must implementations provide additional guarantees about state implicitly included in memory objects for those memory objects that may be exported?

**RESOLVED:** Implementations must ensure that sharing memory objects does not transfer any information between the exporting and importing instances and APIs other than that required to share the data contained in the memory objects explicitly shared. As specific examples, data from previously freed memory objects that used the same underlying physical memory, and data from memory objects using adjacent physical memory must not be visible to applications importing an exported memory object.

9) Must implementations validate external handles the application provides as inputs to memory import operations?

**RESOLVED:** Implementations must return an error to the application if the provided memory handle cannot be used to complete the requested import operation. However, implementations need not validate handles are of the exact type specified by the application.

**Version History**

- Revision 1, 2016-10-20 (James Jones)
  - Initial version

**VK_KHR_external_memory_capabilities**

**Name String**

- **VK_KHR_external_memory_capabilities**
Extension Type
  Instance extension

Registered Extension Number
  72

Revision
  1

Extension and Version Dependencies
  - Requires Vulkan 1.0
  - Requires VK_KHR_get_physical_device_properties2

Deprecation state
  - Promoted to Vulkan 1.1

Contact
  - James Jones cubanismo

Other Extension Metadata

Last Modified Date
  2016-10-17

IP Status
  No known IP claims.

Interactions and External Dependencies
  - Interacts with VK_KHR_dedicated_allocation.
  - Interacts with VK_NV_dedicated_allocation.
  - Promoted to Vulkan 1.1 Core

Contributors
  - Ian Elliot, Google
  - Jesse Hall, Google
  - James Jones, NVIDIA

Description

An application may wish to reference device memory in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension provides a set of capability queries and handle definitions that allow an application to determine what types of “external” memory handles an implementation supports for a given set of use cases.
Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- `vkGetPhysicalDeviceExternalBufferPropertiesKHR`

New Structures

- `VkExternalBufferPropertiesKHR`
- `VkExternalMemoryPropertiesKHR`
- `VkPhysicalDeviceExternalBufferInfoKHR`
- Extending `VkImageFormatProperties2`:
  - `VkExternalImageFormatPropertiesKHR`
- Extending `VkPhysicalDeviceImageFormatInfo2`:
  - `VkPhysicalDeviceExternalImageFormatInfoKHR`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDeviceIDPropertiesKHR`

New Enums

- `VkExternalMemoryFeatureFlagBitsKHR`
- `VkExternalMemoryHandleTypeFlagBitsKHR`

New Bitmasks

- `VkExternalMemoryFeatureFlagsKHR`
- `VkExternalMemoryHandleTypeFlagsKHR`

New Enum Constants

- `VK_KHR_EXTERNAL_MEMORY_CAPABILITIES_EXTENSION_NAME`
- `VK_KHR_EXTERNAL_MEMORY_CAPABILITIES_SPEC_VERSION`
- `VK_LUID_SIZE_KHR`
- Extending `VkExternalMemoryFeatureFlagBits`:
  - `VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_KHR`
  - `VK_EXTERNAL_MEMORY_FEATUREEXPORTABLE_BIT_KHR`
  - `VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_KHR`
- Extending `VkExternalMemoryHandleTypeFlagBits`:
  - `VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_TEXTURE_BIT_KHR`
Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_EXTERNAL_BUFFER_PROPERTIES_KHR`
- `VK_STRUCTURE_TYPE_EXTERNAL_IMAGE_FORMAT_PROPERTIES_KHR`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_BUFFER_INFO_KHR`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_IMAGE_FORMAT_INFO_KHR`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHR`

**Issues**

1) Why do so many external memory capabilities need to be queried on a per-memory-handle-type basis?

**PROPOSED RESOLUTION:** This is because some handle types are based on OS-native objects that have far more limited capabilities than the very generic Vulkan memory objects. Not all memory handle types can name memory objects that support 3D images, for example. Some handle types cannot even support the deferred image and memory binding behavior of Vulkan and require specifying the image when allocating or importing the memory object.

2) Do the `VkExternalImageFormatPropertiesKHR` and `VkExternalBufferPropertiesKHR` structs need to include a list of memory type bits that support the given handle type?

**PROPOSED RESOLUTION:** No. The memory types that don't support the handle types will simply be filtered out of the results returned by `vkGetImageMemoryRequirements` and `vkGetBufferMemoryRequirements` when a set of handle types was specified at image or buffer creation time.

3) Should the non-opaque handle types be moved to their own extension?

**PROPOSED RESOLUTION:** Perhaps. However, defining the handle type bits does very little and does not require any platform-specific types on its own, and it's easier to maintain the bitfield values in a single extension for now. Presumably more handle types could be added by separate extensions though, and it would be midly weird to have some platform-specific ones defined in the core spec and some in extensions.

4) Do we need a `D3D11_TILEPOOL` type?

**PROPOSED RESOLUTION:** No. This is technically possible, but the synchronization is awkward. D3D11 surfaces must be synchronized using shared mutexes, and these synchronization primitives are shared by the entire memory object, so D3D11 shared allocations divided among multiple
buffer and image bindings may be difficult to synchronize.

5) Should the Windows 7-compatible handle types be named “KMT” handles or “GLOBAL_SHARE” handles?

**PROPOSED RESOLUTION:** KMT, simply because it is more concise.

6) How do applications identify compatible devices and drivers across instance, process, and API boundaries when sharing memory?

**PROPOSED RESOLUTION:** New device properties are exposed that allow applications to correctly correlate devices and drivers. A device and driver UUID that must both match to ensure sharing compatibility between two Vulkan instances, or a Vulkan instance and an extensible external API are added. To allow correlating with Direct3D devices, a device LUID is added that corresponds to a DXGI adapter LUID. A driver ID is not needed for Direct3D because mismatched driver component versions are not currently supported on the Windows OS. Should support for such configurations be introduced at the OS level, further Vulkan extensions would be needed to correlate userspace component builds.

**Version History**

- Revision 1, 2016-10-17 (James Jones)
  - Initial version

**VK_KHR_external_semaphore**

**Name String**

VK_KHR_external_semaphore

**Extension Type**

Device extension

**Registered Extension Number**

78

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires VK_KHR_external_semaphore_capabilities

**Deprecation state**

- Promoted to Vulkan 1.1

**Contact**

- James Jones cubanismo
Other Extension Metadata

Last Modified Date
2016-10-21

IP Status
No known IP claims.

Interactions and External Dependencies
• Promoted to Vulkan 1.1 Core

Contributors
• Jason Ekstrand, Intel
• Jesse Hall, Google
• Tobias Hector, Imagination Technologies
• James Jones, NVIDIA
• Jeff Juliano, NVIDIA
• Matthew Netsch, Qualcomm Technologies, Inc.
• Ray Smith, ARM
• Chad Versace, Google

Description
An application using external memory may wish to synchronize access to that memory using semaphores. This extension enables an application to create semaphores from which non-Vulkan handles that reference the underlying synchronization primitive can be exported.

Promotion to Vulkan 1.1
All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures
• Extending VkSemaphoreCreateInfo:
  ◦ VkExportSemaphoreCreateInfoKHR

New Enums
• VkSemaphoreImportFlagBitsKHR

New Bitmasks
• VkSemaphoreImportFlagsKHR
New Enum Constants

- VK_KHR_EXTERNAL_SEMAPHORE_EXTENSION_NAME
- VK_KHR_EXTERNAL_SEMAPHORE_SPEC_VERSION
- Extending VkSemaphoreImportFlagBits:
  - VK_SEMAPHORE_IMPORT_TEMPORARY_BIT_KHR
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHR

Issues

1) Should there be restrictions on what side effects can occur when waiting on imported semaphores that are in an invalid state?

**RESOLVED:** Yes. Normally, validating such state would be the responsibility of the application, and the implementation would be free to enter an undefined state if valid usage rules were violated. However, this could cause security concerns when using imported semaphores, as it would require the importing application to trust the exporting application to ensure the state is valid. Requiring this level of trust is undesirable for many potential use cases.

2) Must implementations validate external handles the application provides as input to semaphore state import operations?

**RESOLVED:** Implementations must return an error to the application if the provided semaphore state handle cannot be used to complete the requested import operation. However, implementations need not validate handles are of the exact type specified by the application.

Version History

- Revision 1, 2016-10-21 (James Jones)
  - Initial revision

**VK_KHR_external_semaphore_capabilities**

Name String

VK_KHR_external_semaphore_capabilities

Extension Type

Instance extension

Registered Extension Number

77

Revision

1
Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2

Deprecation state

• Promoted to Vulkan 1.1

Contact

• James Jones

Other Extension Metadata

Last Modified Date

2016-10-20

IP Status

No known IP claims.

Interactions and External Dependencies

• Promoted to Vulkan 1.1 Core

Contributors

• Jesse Hall, Google
• James Jones, NVIDIA
• Jeff Juliano, NVIDIA

Description

An application may wish to reference device semaphores in multiple Vulkan logical devices or instances, in multiple processes, and/or in multiple APIs. This extension provides a set of capability queries and handle definitions that allow an application to determine what types of “external” semaphore handles an implementation supports for a given set of use cases.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

• vkGetPhysicalDeviceExternalSemaphorePropertiesKHR

New Structures

• VkExternalSemaphorePropertiesKHR
• VkPhysicalDeviceExternalSemaphoreInfoKHR
• Extending VkPhysicalDeviceProperties2:
  ◦ VkPhysicalDeviceIDPropertiesKHR

New Enums

• VkExternalSemaphoreFeatureFlagBitsKHR
• VkExternalSemaphoreHandleTypeFlagBitsKHR

New Bitmasks

• VkExternalSemaphoreFeatureFlagsKHR
• VkExternalSemaphoreHandleTypeFlagsKHR

New Enum Constants

• VK_KHR_EXTERNAL_SEMAPHORE_CAPABILITIES_EXTENSION_NAME
• VK_KHR_EXTERNAL_SEMAPHORE_CAPABILITIES_SPEC_VERSION
• VK_LUID_SIZE_KHR
• Extending VkExternalSemaphoreFeatureFlagBits:
  ◦ VK_EXTERNAL_SEMAPHORE_FEATURE_EXPORTABLE_BIT_KHR
  ◦ VK_EXTERNAL_SEMAPHORE_FEATURE_IMPORTABLE_BIT_KHR
• Extending VkExternalSemaphoreHandleTypeFlagBits:
  ◦ VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_D3D12_FENCE_BIT_KHR
  ◦ VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT_KHR
  ◦ VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR
  ◦ VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_KHR
  ◦ VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_SYNC_FD_BIT_KHR
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_EXTERNAL_SEMAPHORE_PROPERTIES_KHR
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_EXTERNAL_SEMAPHORE_INFO_KHR
  ◦ VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_ID_PROPERTIES_KHR

Version History

• Revision 1, 2016-10-20 (James Jones)
  ◦ Initial revision

VK_KHR_get_memory_requirements2

Name String
  VK_KHR_get_memory_requirements2
Extension Type
  Device extension

Registered Extension Number
  147

Revision
  1

Extension and Version Dependencies
  • Requires Vulkan 1.0

Deprecation state
  • Promoted to Vulkan 1.1

Contact
  • Jason Ekstrand <jekstrand>

Other Extension Metadata

Last Modified Date
  2017-09-05

IP Status
  No known IP claims.

Interactions and External Dependencies
  • Promoted to Vulkan 1.1 Core

Contributors
  • Jason Ekstrand, Intel
  • Jeff Bolz, NVIDIA
  • Jesse Hall, Google

Description

This extension provides new entry points to query memory requirements of images and buffers in a way that can be easily extended by other extensions, without introducing any further entry points. The Vulkan 1.0 VkMemoryRequirements and VkSparseImageMemoryRequirements structures do not include sType and pNext members. This extension wraps them in new structures with these members, so an application can query a chain of memory requirements structures by constructing the chain and letting the implementation fill them in. A new command is added for each vkGet*MemoryRequirements command in core Vulkan 1.0.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The
original type, enum and command names are still available as aliases of the core functionality.

New Commands

- vkGetBufferMemoryRequirements2KHR
- vkGetImageMemoryRequirements2KHR
- vkGetImageSparseMemoryRequirements2KHR

New Structures

- VkBufferMemoryRequirementsInfo2KHR
- VkImageMemoryRequirementsInfo2KHR
- VkImageSparseMemoryRequirementsInfo2KHR
- VkMemoryRequirements2KHR
- VkSparseImageMemoryRequirements2KHR

New Enum Constants

- VK_KHR_GET_MEMORY_REQUIREMENTS_2_EXTENSION_NAME
- VK_KHR_GET_MEMORY_REQUIREMENTS_2_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_BUFFER_MEMORY_REQUIREMENTS_INFO_2_KHR
  - VK_STRUCTURE_TYPE_IMAGE_MEMORY_REQUIREMENTS_INFO_2_KHR
  - VK_STRUCTURE_TYPE_IMAGE_SPARSE_MEMORY_REQUIREMENTS_INFO_2_KHR
  - VK_STRUCTURE_TYPE_MEMORY_REQUIREMENTS_2_KHR
  - VK_STRUCTURE_TYPE_SPARSE_IMAGE_MEMORY_REQUIREMENTS_2_KHR

Version History

- Revision 1, 2017-03-23 (Jason Ekstrand)
  - Internal revisions

VK_KHR_get_physical_device_properties2

Name String

VK_KHR_get_physical_device_properties2

Extension Type

Instance extension

Registered Extension Number

60
This extension provides new entry points to query device features, device properties, and format properties in a way that can be easily extended by other extensions, without introducing any further entry points. The Vulkan 1.0 feature/limit/formatproperty structures do not include sType/pNext members. This extension wraps them in new structures with sType/pNext members, so an application can query a chain of feature/limit/formatproperty structures by constructing the chain and letting the implementation fill them in. A new command is added for each vkGetPhysicalDevice* command in core Vulkan 1.0. The new feature structure (and a pNext chain of extending structures) can also be passed in to device creation to enable features.

This extension also allows applications to use the physical-device components of device extensions before vkCreateDevice is called.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.
New Commands

- vkGetPhysicalDeviceFeatures2KHR
- vkGetPhysicalDeviceFormatProperties2KHR
- vkGetPhysicalDeviceImageFormatProperties2KHR
- vkGetPhysicalDeviceMemoryProperties2KHR
- vkGetPhysicalDeviceProperties2KHR
- vkGetPhysicalDeviceQueueFamilyProperties2KHR
- vkGetPhysicalDeviceSparseImageFormatProperties2KHR

New Structures

- VkFormatProperties2KHR
- VkImageFormatProperties2KHR
- VkPhysicalDeviceImageFormatInfo2KHR
- VkPhysicalDeviceMemoryProperties2KHR
- VkPhysicalDeviceProperties2KHR
- VkPhysicalDeviceSparseImageFormatInfo2KHR
- VkQueueFamilyProperties2KHR
- VkSparseImageFormatProperties2KHR

- Extending VkDeviceCreateInfo:
  - VkPhysicalDeviceFeatures2KHR

New Enum Constants

- VK_KHR_GET_PHYSICAL_DEVICE_PROPERTIES_2_EXTENSION_NAME
- VK_KHR_GET_PHYSICAL_DEVICE_PROPERTIES_2_SPEC_VERSION

- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_FORMAT_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_IMAGE_FORMAT_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_IMAGE_FORMAT_INFO_2_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MEMORY_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SPARSE_IMAGE_FORMAT_INFO_2_KHR
  - VK_STRUCTURE_TYPE_QUEUE_FAMILY_PROPERTIES_2_KHR
  - VK_STRUCTURE_TYPE_SPARSE_IMAGE_FORMAT_PROPERTIES_2_KHR
// Get features with a hypothetical future extension.
VkHypotheticalExtensionFeaturesKHR hypotheticalFeatures =
{
    VK_STRUCTURE_TYPE_HYPOTHETICAL_FEATURES_KHR, // sType
    NULL, // pNext
};

VkPhysicalDeviceFeatures2KHR features =
{
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR, // sType
    &hypotheticalFeatures, // pNext
};

// After this call, features and hypotheticalFeatures have been filled out.
vkGetPhysicalDeviceFeatures2KHR(physicalDevice, &features);

// Properties/limits can be chained and queried similarly.

// Enable some features:
VkHypotheticalExtensionFeaturesKHR enabledHypotheticalFeatures =
{
    VK_STRUCTURE_TYPE_HYPOTHETICAL_FEATURES_KHR, // sType
    NULL, // pNext
};

VkPhysicalDeviceFeatures2KHR enabledFeatures =
{
    VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_FEATURES_2_KHR, // sType
    &enabledHypotheticalFeatures, // pNext
};

enabledFeatures.features.xyz = VK_TRUE;
enabledHypotheticalFeatures.abc = VK_TRUE;

VkDeviceCreateInfo deviceCreateInfo =
{
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO, // sType
    &enabledFeatures, // pNext
    ... // Other arguments
    NULL, // pEnabledFeatures
};

VkDevice device;
vkCreateDevice(physicalDevice, &deviceCreateInfo, NULL, &device);
Version History

- Revision 1, 2016-09-12 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2016-11-02 (Ian Elliott)
  - Added ability for applications to use the physical-device components of device extensions before vkCreateDevice is called.

VK_KHR_maintenance1

Name String

VK_KHR_maintenance1

Extension Type

Device extension

Registered Extension Number

70

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Promoted to Vulkan 1.1

Contact

- Piers Daniell @pdaniell-nv

Other Extension Metadata

Last Modified Date

2018-03-13

Interactions and External Dependencies

- Promoted to Vulkan 1.1 Core

Contributors

- Dan Ginsburg, Valve
- Daniel Koch, NVIDIA
- Daniel Rakos, AMD
- Jan-Harald Fredriksen, ARM
- Jason Ekstrand, Intel
Description

VK_KHR_maintenance1 adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release.

The new features are as follows:

- Allow 2D and 2D array image views to be created from 3D images, which can then be used as color framebuffer attachments. This allows applications to render to slices of a 3D image.

- Support `vkCmdCopyImage` between 2D array layers and 3D slices. This extension allows copying from layers of a 2D array image to slices of a 3D image and vice versa.

- Allow negative height to be specified in the `VkViewport::height` field to perform y-inversion of the clip-space to framebuffer-space transform. This allows apps to avoid having to use `gl_Position.y = -gl_Position.y` in shaders also targeting other APIs.

- Allow implementations to express support for doing just transfers and clears of image formats that they otherwise support no other format features for. This is done by adding new format feature flags `VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR` and `VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR`.

- Support `vkCmdFillBuffer` on transfer-only queues. Previously `vkCmdFillBuffer` was defined to only work on command buffers allocated from command pools which support graphics or compute queues. It is now allowed on queues that just support transfer operations.

- Fix the inconsistency of how error conditions are returned between the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` functions and the `vkAllocateDescriptorSets` and `vkAllocateCommandBuffers` functions.

- Add new `VK_ERROR_OUT_OF_POOL_MEMORY_KHR` error so implementations can give a more precise reason for `vkAllocateDescriptorSets` failures.

- Add a new command `vkTrimCommandPoolKHR` which gives the implementation an opportunity to release any unused command pool memory back to the system.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.
New Commands

- \texttt{vkTrimCommandPoolKHR}

New Bitmasks

- \texttt{VkCommandPoolTrimFlagsKHR}

New Enum Constants

- \texttt{VK_KHR_MAINTENANCE1_EXTENSION_NAME}
- \texttt{VK_KHR_MAINTENANCE1_SPEC_VERSION}
- Extending \texttt{VkFormatFeatureFlagBits}:
  - \texttt{VK_FORMAT_FEATURE_TRANSFER_DST_BIT_KHR}
  - \texttt{VK_FORMAT_FEATURE_TRANSFER_SRC_BIT_KHR}
- Extending \texttt{VkImageCreateFlagBits}:
  - \texttt{VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR}
- Extending \texttt{VkResult}:
  - \texttt{VK_ERROR_OUT_OF_POOL_MEMORY_KHR}

Issues

1. Are viewports with zero height allowed?

   \textbf{RESOLVED}: Yes, although they have low utility.

Version History

- Revision 1, 2016-10-26 (Piers Daniell)
  - Internal revisions
- Revision 2, 2018-03-13 (Jon Leech)
  - Add issue for zero-height viewports

\textbf{VK\_KHR\_maintenance2}

Name String

- \texttt{VK\_KHR\_maintenance2}

Extension Type

- Device extension

Registered Extension Number

- 118
Description

`VK_KHR_maintenance2` adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release.

The new features are as follows:

- Allow the application to specify which aspect of an input attachment might be read for a given subpass.
- Allow implementations to express the clipping behavior of points.
- Allow creating images with usage flags that may not be supported for the base image’s format, but are supported for image views of the image that have a different but compatible format.
- Allow creating uncompressed image views of compressed images.
- Allow the application to select between an upper-left and lower-left origin for the tessellation...
domain space.

- Adds two new image layouts for depth stencil images to allow either the depth or stencil aspect to be read-only while the other aspect is writable.

**Input Attachment Specification**

Input attachment specification allows an application to specify which aspect of a multi-aspect image (e.g. a combined depth stencil format) will be accessed via a `subpassLoad` operation.

On some implementations there *may* be a performance penalty if the implementation does not know (at `vkCreateRenderPass` time) which aspect(s) of multi-aspect images *can* be accessed as input attachments.

**Promotion to Vulkan 1.1**

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

**New Structures**

- `VkInputAttachmentAspectReferenceKHR`
- Extending `VkImageViewCreateInfo`:
  - `VkImageViewUsageCreateInfoKHR`
- Extending `VkPhysicalDeviceProperties2`:
  - `VkPhysicalDevicePointClippingPropertiesKHR`
- Extending `VkPipelineTessellationStateCreateInfo`:
  - `VkPipelineTessellationDomainOriginStateCreateInfoKHR`
- Extending `VkRenderPassCreateInfo`:
  - `VkRenderPassInputAttachmentAspectCreateInfoKHR`

**New Enums**

- `VkPointClippingBehaviorKHR`
- `VkTessellationDomainOriginKHR`

**New Enum Constants**

- `VK_KHR_MAINTENANCE2_EXTENSION_NAME`
- `VK_KHR_MAINTENANCE2_SPEC_VERSION`
- Extending `VkImageCreateFlagBits`:
  - `VK_IMAGE_CREATE_BLOCK_TEXEL_VIEW_COMPATIBLE_BIT_KHR`
  - `VK_IMAGE_CREATE_EXTENDED_USAGE_BIT_KHR`
- Extending `VkImageLayout`: 3036 | Appendix E: Layers & Extensions (Informative)
• Extending `VkPointClippingBehavior`:
  ◦ `VK_POINT_CLIPPING_BEHAVIOR_ALL_CLIP_PLANES_KHR`
  ◦ `VK_POINT_CLIPPING_BEHAVIOR_USER_CLIP_PLANES_ONLY_KHR`

• Extending `VkStructureType`:
  ◦ `VK_STRUCTURE_TYPE_IMAGE_VIEW_USAGE_CREATE_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_POINT_CLIPPING_PROPERTIES_KHR`
  ◦ `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_DOMAIN_ORIGIN_STATE_CREATE_INFO_KHR`
  ◦ `VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO_KHR`

• Extending `VkTessellationDomainOrigin`:
  ◦ `VK_TESSELLATION_DOMAIN_ORIGIN_LOWER_LEFT_KHR`
  ◦ `VK_TESSELLATION_DOMAIN_ORIGIN_UPPER_LEFT_KHR`

**Input Attachment Specification Example**

Consider the case where a render pass has two subpasses and two attachments.

Attachment 0 has the format `VK_FORMAT_D24_UNORM_S8_UINT`, attachment 1 has some color format.

Subpass 0 writes to attachment 0, subpass 1 reads only the depth information from attachment 0 (using `inputAttachmentRead`) and writes to attachment 1.
VkInputAttachmentAspectReferenceKHR references[] = {
    {
        .subpass = 1,
        .inputAttachmentIndex = 0,
        .aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT
    }
};

VkRenderPassInputAttachmentAspectCreateInfoKHR specifyAspects = {
    .sType = VK_STRUCTURE_TYPE_RENDER_PASS_INPUT_ATTACHMENT_ASPECT_CREATE_INFO_KHR,
    .pNext = NULL,
    .aspectReferenceCount = 1,
    .pAspectReferences = references
};

VkRenderPassCreateInfo createInfo = {
    ...,
    .pNext = &specifyAspects,
    ...,
};

vkCreateRenderPass(...);

Issues

1) What is the default tessellation domain origin?

RESOLVED: Vulkan 1.0 originally inadvertently documented a lower-left origin, but the conformance tests and all implementations implemented an upper-left origin. This extension adds a control to select between lower-left (for compatibility with OpenGL) and upper-left, and we retroactively fix unextended Vulkan to have a default of an upper-left origin.

Version History

• Revision 1, 2017-04-28

VK_KHR_maintenance3

Name String

VK_KHR_maintenance3

Extension Type

Device extension

Registered Extension Number

169
Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires `VK_KHR_get_physical_device_properties2`

Deprecation state
• Promoted to Vulkan 1.1

Contact
• Jeff Bolz (@jeffbolznv)

Other Extension Metadata

Last Modified Date
2017-09-05

Interactions and External Dependencies
• Promoted to Vulkan 1.1 Core

Contributors
• Jeff Bolz, NVIDIA

Description

`VK_KHR_maintenance3` adds a collection of minor features that were intentionally left out or overlooked from the original Vulkan 1.0 release.

The new features are as follows:

• A limit on the maximum number of descriptors that are supported in a single descriptor set layout. Some implementations have a limit on the total size of descriptors in a set, which cannot be expressed in terms of the limits in Vulkan 1.0.

• A limit on the maximum size of a single memory allocation. Some platforms have kernel interfaces that limit the maximum size of an allocation.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Commands
• `vkGetDescriptorSetLayoutSupportKHR`
New Structures

- VkDescriptorSetLayoutSupportKHR
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceMaintenance3PropertiesKHR

New Enum Constants

- VK_KHR_MAINTENANCE3_EXTENSION_NAME
- VK_KHR_MAINTENANCE3_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_SUPPORT_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MAINTENANCE_3_PROPERTIES_KHR

Version History

- Revision 1, 2017-08-22

VK_KHR_multiview

Name String
  VK_KHR_multiview

Extension Type
  Device extension

Registered Extension Number
  54

Revision
  1

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_get_physical_device_properties2

Deprecation state

- Promoted to Vulkan 1.1

Contact

- Jeff Bolz @jeffbolz

Other Extension Metadata
IP Status
No known IP claims.

Interactions and External Dependencies
- Promoted to Vulkan 1.1 Core
- This extension requires SPV_KHR_multiview
- This extension provides API support for GL_EXT_multiview

Contributors
- Jeff Bolz, NVIDIA

Description
This extension has the same goal as the OpenGL ES GL_OVR_multiview extension. Multiview is a rendering technique originally designed for VR where it is more efficient to record a single set of commands to be executed with slightly different behavior for each “view”.

It includes a concise way to declare a render pass with multiple views, and gives implementations freedom to render the views in the most efficient way possible. This is done with a multiview configuration specified during render pass creation with the VkRenderPassMultiviewCreateInfo passed into VkRenderPassCreateInfo::pNext.

This extension enables the use of the SPV_KHR_multiview shader extension, which adds a new ViewIndex built-in type that allows shaders to control what to do for each view. If using GLSL there is also the GL_EXT_multiview extension that introduces a highp int gl_ViewIndex; built-in variable for vertex, tessellation, geometry, and fragment shaders.

Promotion to Vulkan 1.1
All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Structures
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceMultiviewFeaturesKHR
- Extending VkPhysicalDeviceProperties2:
  - VkPhysicalDeviceMultiviewPropertiesKHR
- Extending VkRenderPassCreateInfo:
  - VkRenderPassMultiviewCreateInfoKHR
New Enum Constants

- VK_KHR_MULTIVIEW_EXTENSION_NAME
- VK_KHR_MULTIVIEW_SPEC_VERSION

Extending VkDependencyFlagBits:
  - VK_DEPENDENCY_VIEW_LOCAL_BIT_KHR

Extending VkStructureType:
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_FEATURES_KHR
  - VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_MULTIVIEW_PROPERTIES_KHR
  - VK_STRUCTURE_TYPE_RENDER_PASS_MULTIVIEW_CREATE_INFO_KHR

New Built-In Variables

- ViewIndex

New SPIR-V Capabilities

- MultiView

Version History

- Revision 1, 2016-10-28 (Jeff Bolz)
  - Internal revisions

VK_KHR_relaxed_block_layout

Name String
  VK_KHR_relaxed_block_layout

Extension Type
  Device extension

Registered Extension Number
  145

Revision
  1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Promoted to Vulkan 1.1
Description

The VK_KHR_relaxed_block_layout extension allows implementations to indicate they can support more variation in block Offset decorations. For example, placing a vector of three floats at an offset of $16 \times N + 4$.

See Offset and Stride Assignment for details.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. The original type, enum and command names are still available as aliases of the core functionality.

New Enum Constants

- VK_KHR_RELAXED_BLOCK_LAYOUT_EXTENSION_NAME
- VK_KHR_RELAXED_BLOCK_LAYOUT_SPEC_VERSION

Version History

- Revision 1, 2017-03-26 (JohnK)

VK_KHR_sampler_ycbcr_conversion

Name String

VK_KHR_sampler_ycbcr_conversion

Extension Type

Device extension
Registered Extension Number
157

Revision
14

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_KHR_maintenance1
• Requires VK_KHR_bind_memory2
• Requires VK_KHR_get_memory_requirements2
• Requires VK_KHR_get_physical_device_properties2

Deprecation state
• Promoted to Vulkan 1.1

Contact
• Andrew Garrard fluppeteer

Other Extension Metadata

Last Modified Date
2017-08-11

IP Status
No known IP claims.

Interactions and External Dependencies
• Promoted to Vulkan 1.1 Core

Contributors
• Andrew Garrard, Samsung Electronics
• Tobias Hector, Imagination Technologies
• James Jones, NVIDIA
• Daniel Koch, NVIDIA
• Daniel Rakos, AMD
• Romain Guy, Google
• Jesse Hall, Google
• Tom Cooksey, ARM Ltd
• Jeff Leger, Qualcomm Technologies, Inc
• Jan-Harald Fredriksen, ARM Ltd
• Jan Outters, Samsung Electronics
Description

The use of Y′C₈C₉ sampler conversion is an area in 3D graphics not used by most Vulkan developers. It's mainly used for processing inputs from video decoders and cameras. The use of the extension assumes basic knowledge of Y′C₈C₉ concepts.

This extension provides the ability to perform specified color space conversions during texture sampling operations for the Y′C₈C₉ color space natively. It also adds a selection of multi-planar formats, image aspect plane, and the ability to bind memory to the planes of an image collectively or separately.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted. However, if Vulkan 1.1 is supported and this extension is not, the samplerYcbcrConversion capability is optional. The original type, enum and command names are still available as aliases of the core functionality.

New Object Types

- VkSamplerYcbcrConversionKHR

New Commands

- vkCreateSamplerYcbcrConversionKHR
- vkDestroySamplerYcbcrConversionKHR

New Structures

- VkSamplerYcbcrConversionCreateInfoKHR
- Extending VkBindImageMemoryInfo:
  - VkBindImagePlaneMemoryInfoKHR
- Extending VkImageFormatProperties2:
  - VkSamplerYcbcrConversionImageFormatPropertiesKHR
- Extending VkImageMemoryRequirementsInfo2:
  - VkImagePlaneMemoryRequirementsInfoKHR
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceSamplerYcbcrConversionFeaturesKHR
Extending `VkSamplerCreateInfo`, `VkImageViewCreateInfo`:
- `VkSamplerYcbcrConversionInfoKHR`

**New Enums**
- `VkChromaLocationKHR`
- `VkSamplerYcbcrModelConversionKHR`
- `VkSamplerYcbcrRangeKHR`

**New Enum Constants**
- `VK_KHR_SAMPLER_YCBCR_CONVERSION_EXTENSION_NAME`
- `VK_KHR_SAMPLER_YCBCR_CONVERSION_SPEC_VERSION`

Extending `VkChromaLocation`:
- `VK_CHROMA_LOCATION_COSITED_EVEN_KHR`
- `VK_CHROMA_LOCATION_MIDPOINT_KHR`

Extending `VkDebugReportObjectTypeEXT`:
- `VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_KHR_EXT`

Extending `VkFormat`:
- `VK_FORMAT_B10X6G10X6R10X6G10X6_422_UNORM_4PACK16_KHR`
- `VK_FORMAT_B12X4G12X4R12X4G12X4_422_UNORM_4PACK16_KHR`
- `VK_FORMAT_B16G16R16G16_422_UNORM_KHR`
- `VK_FORMAT_B8G8R8G8_422_UNORM_KHR`
- `VK_FORMAT_G10X6B10X6G10X6R10X6_422_UNORM_4PACK16_KHR`
- `VK_FORMAT_G10X6_B10X6R10X6_2PLANE_420_UNORM_3PACK16_KHR`
- `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_420_UNORM_3PACK16_KHR`
- `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_422_UNORM_3PACK16_KHR`
- `VK_FORMAT_G10X6_B10X6_R10X6_3PLANE_444_UNORM_3PACK16_KHR`
- `VK_FORMAT_G12X4B12X4G12X4R12X4_422_UNORM_4PACK16_KHR`
- `VK_FORMAT_G12X4_B12X4R12X4_2PLANE_420_UNORM_3PACK16_KHR`
- `VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_420_UNORM_3PACK16_KHR`
- `VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_422_UNORM_3PACK16_KHR`
- `VK_FORMAT_G12X4_B12X4_R12X4_3PLANE_444_UNORM_3PACK16_KHR`
- `VK_FORMAT_G16B16G16R16_422_UNORM_KHR`
- `VK_FORMAT_G16_B16R16_2PLANE_420_UNORM_KHR`
• Extending \textit{VkFormatFeatureFlagBits}:
  - VK_FORMAT_FEATURE_COSITED_CHROMA_SAMPLES_BIT_KHR
  - VK_FORMAT_FEATURE_DISJOINT_BIT_KHR
  - VK_FORMAT_FEATURE_MIDPOINT_CHROMA_SAMPLES_BIT_KHR
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_BIT_KHR
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_CHROMA_RECONSTRUCTION_EXPLICIT_FORCEABLE_BIT_KHR
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_LINEAR_FILTER_BIT_KHR
  - VK_FORMAT_FEATURE_SAMPLED_IMAGE_YCBCR_CONVERSION_SEPARATE_RECONSTRUCTION_FILTER_BIT_KHR

• Extending \textit{VkImageAspectFlagBits}:
  - VK_IMAGE_ASPECT_PLANE_0_BIT_KHR
  - VK_IMAGE_ASPECT_PLANE_1_BIT_KHR
  - VK_IMAGE_ASPECT_PLANE_2_BIT_KHR

• Extending \textit{VkImageCreateFlagBits}:
  - VK_IMAGE_CREATE_DISJOINT_BIT_KHR

• Extending \textit{VkObjectType}:
  - VK_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_KHR

• Extending \textit{VkSamplerYcbcrModelConversion}:
  - VK_SAMPLER_YCBCR_MODEL_CONVERSION_RGB_IDENTITY_KHR
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_2020_KHR`
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_601_KHR`
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_709_KHR`
- `VK_SAMPLER_YCBCR_MODEL_CONVERSION_YCBCR_IDENTITY_KHR`

**Extending `VkSamplerYcbcrRange`:**
- `VK_SAMPLER_YCBCR_RANGE_ITU_FULL_KHR`
- `VK_SAMPLER_YCBCR_RANGE_ITU_NARROW_KHR`

**Extending `VkStructureType`:**
- `VK_STRUCTURE_TYPE_BIND_IMAGE_PLANE_MEMORY_INFO_KHR`
- `VK_STRUCTURE_TYPE_IMAGE_PLANE_MEMORY_REQUIREMENTS_INFO_KHR`
- `VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_SAMPLER_YCBCR_CONVERSION_FEATURES_KHR`
- `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_CREATE_INFO_KHR`
- `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_IMAGE_FORMAT_PROPERTIES_KHR`
- `VK_STRUCTURE_TYPE_SAMPLER_YCBCR_CONVERSION_INFO_KHR`

If `VK_EXT_debug_report` is supported:

**Extending `VkDebugReportObjectTypeEXT`:**
- `VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_EXT`

**Version History**

- **Revision 1, 2017-01-24 (Andrew Garrard)**
  - Initial draft
- **Revision 2, 2017-01-25 (Andrew Garrard)**
  - After initial feedback
- **Revision 3, 2017-01-27 (Andrew Garrard)**
  - Higher bit depth formats, renaming, swizzle
- **Revision 4, 2017-02-22 (Andrew Garrard)**
  - Added query function, formats as RGB, clarifications
- **Revision 5, 2017-04-?? (Andrew Garrard)**
  - Simplified query and removed output conversions
- **Revision 6, 2017-04-24 (Andrew Garrard)**
  - Tidying, incorporated new image query, restored transfer functions
- **Revision 7, 2017-04-25 (Andrew Garrard)**
  - Added cosited option/midpoint requirement for formats, “bypassConversion”
- **Revision 8, 2017-04-25 (Andrew Garrard)**
- Simplified further
- Revision 9, 2017-04-27 (Andrew Garrard)
  - Disjoint no more
- Revision 10, 2017-04-28 (Andrew Garrard)
  - Restored disjoint
- Revision 11, 2017-04-29 (Andrew Garrard)
  - Now Ycbcr conversion, and KHR
- Revision 12, 2017-06-06 (Andrew Garrard)
  - Added conversion to image view creation
- Revision 13, 2017-07-13 (Andrew Garrard)
  - Allowed cosited-only chroma samples for formats
- Revision 14, 2017-08-11 (Andrew Garrard)
  - Reflected quantization changes in BT.2100-1

**VK_KHR_shader_draw_parameters**

Name String

VK_KHR_shader_draw_parameters

Extension Type

Device extension

Registered Extension Number

64

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Promoted to Vulkan 1.1

Contact

- Daniel Koch dgkoch

Other Extension Metadata

Last Modified Date

2017-09-05
IP Status
No known IP claims.

Interactions and External Dependencies
- This extension requires `SPV_KHR_shader_draw_parameters`
- This extension provides API support for `GL_ARB_shader_draw_parameters`
- Promoted to Vulkan 1.1 Core

Contributors
- Daniel Koch, NVIDIA Corporation
- Jeff Bolz, NVIDIA
- Daniel Rakos, AMD
- Jan-Harald Fredriksen, ARM
- John Kessenich, Google
- Stuart Smith, IMG

Description
This extension adds support for the following SPIR-V extension in Vulkan:

- `SPV_KHR_shader_draw_parameters`

The extension provides access to three additional built-in shader variables in Vulkan:

- `BaseInstance`, which contains the `firstInstance` parameter passed to draw commands,
- `BaseVertex`, which contains the `firstVertex` or `vertexOffset` parameter passed to draw commands, and
- `DrawIndex`, which contains the index of the draw call currently being processed from an indirect draw call.

When using GLSL source-based shader languages, the following variables from `GL_ARB_shader_draw_parameters` can map to these SPIR-V built-in decorations:

- `in int gl_BaseInstanceARB; → BaseInstance,`
- `in int gl_BaseVertexARB; → BaseVertex, and`
- `in int gl_DrawIDARB; → DrawIndex.`

Promotion to Vulkan 1.1
All functionality in this extension is included in core Vulkan 1.1, however a feature bit was added to distinguish whether it is actually available or not.

New Enum Constants
- `VK_KHR_SHADER_DRAW_PARAMETERS_EXTENSION_NAME`
New Built-In Variables

- BaseInstance
- BaseVertex
- DrawIndex

New SPIR-V Capabilities

- DrawParameters

Issues

1) Is this the same functionality as GL_ARB_shader_draw_parameters?

RESOLVED: It’s actually a superset as it also adds in support for arrayed drawing commands.

In GL for GL_ARB_shader_draw_parameters, gl_BaseVertexARB holds the integer value passed to the parameter to the command that resulted in the current shader invocation. In the case where the command has no baseVertex parameter, the value of gl_BaseVertexARB is zero. This means that gl_BaseVertexARB = baseVertex (for glDrawElements commands with baseVertex) or 0. In particular there are no glDrawArrays commands that take a baseVertex parameter.

Now in Vulkan, we have BaseVertex = vertexOffset (for indexed drawing commands) or firstVertex (for arrayed drawing commands), and so Vulkan’s version is really a superset of GL functionality.

Version History

- Revision 1, 2016-10-05 (Daniel Koch)
  - Internal revisions

VK_KHR_storage_buffer_storage_class

Name String

VK_KHR_storage_buffer_storage_class

Extension Type

Device extension

Registered Extension Number

132

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0
This extension adds support for the following SPIR-V extension in Vulkan:

- **SPV_KHR_storage_buffer_storage_class**

This extension provides a new SPIR-V `StorageBuffer` storage class. A `Block`-decorated object in this class is equivalent to a `BufferBlock`-decorated object in the `Uniform` storage class.

### Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1.

### New Enum Constants

- **VK_KHR_STORAGE_BUFFER_STORAGE_CLASS_EXTENSION_NAME**
- **VK_KHR_STORAGE_BUFFER_STORAGE_CLASS_SPEC_VERSION**

### Version History

- Revision 1, 2017-03-23 (Alexander Galazin)
  - Initial draft
VK_KHR_variable_pointers

Name String

VK_KHR_variable_pointers

Extension Type

Device extension

Registered Extension Number

121

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_KHR_get_physical_device_properties2
• Requires VK_KHR_storage_buffer_storage_class

Deprecation state

• Promoted to Vulkan 1.1

Contact

• Jesse Hall [critsec]

Other Extension Metadata

Last Modified Date

2017-09-05

IP Status

No known IP claims.

Interactions and External Dependencies

• This extension requires SPV_KHR_variable_pointers
• Promoted to Vulkan 1.1 Core

Contributors

• John Kessenich, Google
• Neil Henning, Codeplay
• David Neto, Google
• Daniel Koch, Nvidia
• Graeme Leese, Broadcom
• Weifeng Zhang, Qualcomm
• Stephen Clarke, Imagination Technologies
Description

The \texttt{VK_KHR_variable_pointers} extension allows implementations to indicate their level of support for the \texttt{SPV_KHR_variable_pointers} SPIR-V extension. The SPIR-V extension allows shader modules to use invocation-private pointers into uniform and/or storage buffers, where the pointer values can be dynamic and non-uniform.

The \texttt{SPV_KHR_variable_pointers} extension introduces two capabilities. The first, \texttt{VariablePointersStorageBuffer}, must be supported by all implementations of this extension. The second, \texttt{VariablePointers}, is optional.

Promotion to Vulkan 1.1

All functionality in this extension is included in core Vulkan 1.1, with the KHR suffix omitted, however support for the \texttt{variablePointersStorageBuffer} feature is made optional. The original type, enum and command names are still available as aliases of the core functionality.

New Structures

- Extending \texttt{VkPhysicalDeviceFeatures2}, \texttt{VkDeviceCreateInfo}:
  - \texttt{VkPhysicalDeviceVariablePointerFeaturesKHR}
  - \texttt{VkPhysicalDeviceVariablePointersFeaturesKHR}

New Enum Constants

- \texttt{VK_KHR_VARIABLE_POINTERS_EXTENSION_NAME}
- \texttt{VK_KHR_VARIABLE_POINTERS_SPEC_VERSION}

Extending \texttt{VkStructureType}:
  - \texttt{VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_VARIABLE_POINTERS_FEATURES_KHR}

New SPIR-V Capabilities

- \texttt{VariablePointers}
- \texttt{VariablePointersStorageBuffer}

Issues

1) Do we need an optional property for the SPIR-V \texttt{VariablePointersStorageBuffer} capability or should it be mandatory when this extension is advertised?

\textbf{RESOLVED}: Add it as a distinct feature, but make support mandatory. Adding it as a feature makes the extension easier to include in a future core API version. In the extension, the feature is mandatory, so that presence of the extension guarantees some functionality. When included in a
core API version, the feature would be optional.

2) Can support for these capabilities vary between shader stages?

**RESOLVED**: No, if the capability is supported in any stage it must be supported in all stages.

3) Should the capabilities be features or limits?

**RESOLVED**: Features, primarily for consistency with other similar extensions.

### Version History

- Revision 1, 2017-03-14 (Jesse Hall and John Kessenich)
  - Internal revisions

### VK_EXT_buffer_device_address

#### Name String

`VK_EXT_buffer_device_address`

#### Extension Type

Device extension

#### Registered Extension Number

245

#### Revision

2

#### Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires `VK_KHR_get_physical_device_properties2`

#### Deprecation state

- *Deprecated* by `VK_KHR_buffer_device_address` extension
  - Which in turn was *promoted* to Vulkan 1.2

#### Contact

- Jeff Bolz [@jeffbolznv](https://jeffbolznv)

### Other Extension Metadata

#### Last Modified Date

2019-01-06

#### IP Status

No known IP claims.
Interactions and External Dependencies

- This extension requires SPV_EXT_physical_storage_buffer

Contributors

- Jeff Bolz, NVIDIA
- Neil Henning, AMD
- Tobias Hector, AMD
- Jason Ekstrand, Intel
- Baldur Karlsson, Valve

Description

This extension allows the application to query a 64-bit buffer device address value for a buffer, which can be used to access the buffer memory via the PhysicalStorageBufferEXT storage class in the GL_EXT_buffer_reference GLSL extension and SPV_EXT_physical_storage_buffer SPIR-V extension.

It also allows buffer device addresses to be provided by a trace replay tool, so that it matches the address used when the trace was captured.

New Commands

- vkGetBufferDeviceAddressEXT

New Structures

- VkBufferDeviceAddressInfoEXT
- Extending VkBufferCreateInfo:
  - VkBufferDeviceAddressCreateInfoEXT
- Extending VkPhysicalDeviceFeatures2, VkDeviceCreateInfo:
  - VkPhysicalDeviceBufferDeviceAddressFeaturesEXT

New Enum Constants

- VK_EXT_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME
- VK_EXT_BUFFER_DEVICE_ADDRESS_SPEC_VERSION
- Extending VkBufferCreateFlagBits:
  - VK_BUFFER_CREATE_DEVICE_ADDRESS_CAPTURE_REPLAY_BIT_EXT
- Extending VkBufferUsageFlagBits:
  - VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT_EXT
- Extending VkResult:
  - VK_ERROR_INVALID_DEVICE_ADDRESS_EXT
- Extending VkStructureType:
New SPIR-V Capabilities

- PhysicalStorageBufferAddressesEXT

Issues

1) Where is VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_ADDRESS_FEATURES_EXT and VkPhysicalDeviceBufferAddressFeaturesEXT?

RESOLVED: They were renamed as VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_BUFFER_DEVICE_ADDRESS_FEATURES_EXT and VkPhysicalDeviceBufferDeviceAddressFeaturesEXT accordingly for consistency. Even though, the old names can still be found in the generated header files for compatibility.

Version History

- Revision 1, 2018-11-01 (Jeff Bolz)
  - Internal revisions
- Revision 2, 2019-01-06 (Jon Leech)
  - Minor updates to appendix for publication

VK_EXT_debug_marker

Name String

VK_EXT_debug_marker

Extension Type

Device extension

Registered Extension Number

23

Revision

4

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_EXT_debug_report

Deprecation state

- Promoted to VK_EXT_debug_utils extension
Description

The `VK_EXT_debug_marker` extension is a device extension. It introduces concepts of object naming and tagging, for better tracking of Vulkan objects, as well as additional commands for recording annotations of named sections of a workload to aid organization and offline analysis in external tools.

New Commands

- `vkCmdDebugMarkerBeginEXT`
- `vkCmdDebugMarkerEndEXT`
- `vkCmdDebugMarkerInsertEXT`
- `vkDebugMarkerSetObjectNameEXT`
- `vkDebugMarkerSetObjectTagEXT`

New Structures

- `VkDebugMarkerMarkerInfoEXT`
- `VkDebugMarkerObjectNameInfoEXT`
- `VkDebugMarkerObjectTagInfoEXT`

New Enums

- `VkDebugReportObjectTypeEXT`
New Enum Constants

- VK_EXT_DEBUG_MARKER_EXTENSION_NAME
- VK_EXT_DEBUG_MARKER_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT
  - VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT
  - VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_TAG_INFO_EXT

Examples

Example 1

Associate a name with an image, for easier debugging in external tools or with validation layers that can print a friendly name when referring to objects in error messages.

```java
extern VkDevice device;
extern VkImage image;

// Must call extension functions through a function pointer:
PFN_vkDebugMarkerSetObjectNameEXT pfnDebugMarkerSetObjectNameEXT =
    (PFN_vkDebugMarkerSetObjectNameEXT)vkGetDeviceProcAddr(device,
                                                             "vkDebugMarkerSetObjectNameEXT");

// Set a name on the image
const VkDebugMarkerObjectNameInfoEXT imageNameInfo =
    {
        VK_STRUCTURE_TYPE_DEBUG_MARKER_OBJECT_NAME_INFO_EXT, // sType
        NULL, // pNext
        VK_DEBUG_REPORT_OBJECT_TYPE_IMAGE_EXT, // objectType
        (uint64_t)image, // object
        "Brick Diffuse Texture", // pObjectName
    };

pfnDebugMarkerSetObjectNameEXT(device, &imageNameInfo);

// A subsequent error might print:
// Image 'Brick Diffuse Texture' (0xc0dec0dedeadbeef) is used in a
// command buffer with no memory bound to it.
```

Example 2

Annotating regions of a workload with naming information so that offline analysis tools can display a more usable visualisation of the commands submitted.

```java
extern VkDevice device;
extern VkCommandBuffer commandBuffer;
```
// Must call extension functions through a function pointer:
PFN_vkCmdDebugMarkerBeginEXT pfnCmdDebugMarkerBeginEXT = (PFN_vkCmdDebugMarkerBeginEXT)vkGetDeviceProcAddr(device, "vkCmdDebugMarkerBeginEXT");
PFN_vkCmdDebugMarkerEndEXT pfnCmdDebugMarkerEndEXT = (PFN_vkCmdDebugMarkerEndEXT)vkGetDeviceProcAddr(device, "vkCmdDebugMarkerEndEXT");
PFN_vkCmdDebugMarkerInsertEXT pfnCmdDebugMarkerInsertEXT = (PFN_vkCmdDebugMarkerInsertEXT)vkGetDeviceProcAddr(device, "vkCmdDebugMarkerInsertEXT");

// Describe the area being rendered
const VkDebugMarkerMarkerInfoEXT houseMarker = {
    VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT, // sType
    NULL, // pNext
    "Brick House", // pMarkerName
    {1.0f, 0.0f, 0.0f, 1.0f}, // color
};

// Start an annotated group of calls under the 'Brick House' name
pfnCmdDebugMarkerBeginEXT(commandBuffer, &houseMarker);
{
    // A mutable structure for each part being rendered
    VkDebugMarkerMarkerInfoEXT housePartMarker = {
        VK_STRUCTURE_TYPE_DEBUG_MARKER_MARKER_INFO_EXT, // sType
        NULL, // pNext
        NULL, // pMarkerName
        {0.0f, 0.0f, 0.0f, 0.0f}, // color
    };

    // Set the name and insert the marker
    housePartMarker.pMarkerName = "Walls";
pfnCmdDebugMarkerInsertEXT(commandBuffer, &housePartMarker);

    // Insert the drawcall for the walls
    vkCmdDrawIndexed(commandBuffer, 1000, 1, 0, 0, 0);

    // Insert a recursive region for two sets of windows
    housePartMarker.pMarkerName = "Windows";
pfnCmdDebugMarkerBeginEXT(commandBuffer, &housePartMarker);
    {
        vkCmdDrawIndexed(commandBuffer, 75, 6, 1000, 0, 0);
vkCmdDrawIndexed(commandBuffer, 100, 2, 1450, 0, 0);
    }
pfnCmdDebugMarkerEndEXT(commandBuffer);

    housePartMarker.pMarkerName = "Front Door";
pfnCmdDebugMarkerInsertEXT(commandBuffer, &housePartMarker);

    vkCmdDrawIndexed(commandBuffer, 350, 1, 1650, 0, 0);
Issues

1) Should the tag or name for an object be specified using the pNext parameter in the object's Vk*CreateInfo structure?

**RESOLVED**: No. While this fits with other Vulkan patterns and would allow more type safety and future proofing against future objects, it has notable downsides. In particular passing the name at Vk*CreateInfo time does not allow renaming, prevents late binding of naming information, and does not allow naming of implicitly created objects such as queues and swapchain images.

2) Should the command annotation functions vkCmdDebugMarkerBeginEXT and vkCmdDebugMarkerEndEXT support the ability to specify a color?

**RESOLVED**: Yes. The functions have been expanded to take an optional color which can be used at will by implementations consuming the command buffer annotations in their visualisation.

3) Should the functions added in this extension accept an extensible structure as their parameter for a more flexible API, as opposed to direct function parameters? If so, which functions?

**RESOLVED**: Yes. All functions have been modified to take a structure type with extensible pNext pointer, to allow future extensions to add additional annotation information in the same commands.

Version History

- Revision 1, 2016-02-24 (Baldur Karlsson)
  - Initial draft, based on LunarG marker spec
- Revision 2, 2016-02-26 (Baldur Karlsson)
  - Renamed Dbg to DebugMarker in function names
  - Allow markers in secondary command buffers under certain circumstances
  - Minor language tweaks and edits
- Revision 3, 2016-04-23 (Baldur Karlsson)
  - Reorganise spec layout to closer match desired organisation
  - Added optional color to markers (both regions and inserted labels)
  - Changed functions to take extensible structs instead of direct function parameters
- Revision 4, 2017-01-31 (Baldur Karlsson)
- Added explicit dependency on VK_EXT_debug_report
- Moved definition of VkDebugReportObjectTypeEXT to debug report chapter.
- Fixed typo in dates in revision history

**VK_EXT_debug_report**

**Name String**

VK_EXT_debug_report

**Extension Type**

Instance extension

**Registered Extension Number**

12

**Revision**

10

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Deprecation state**

- Deprecated by VK_EXT_debug_utils extension

**Special Use**

- Debugging tools

**Contact**

- Courtney Goeltzenleuchter [courtney-g](https://github.com/courtney-g)

**Other Extension Metadata**

**Last Modified Date**

2020-12-14

**IP Status**

No known IP claims.

**Contributors**

- Courtney Goeltzenleuchter, LunarG
- Dan Ginsburg, Valve
- Jon Ashburn, LunarG
- Mark Lobodzinski, LunarG
Description

Due to the nature of the Vulkan interface, there is very little error information available to the developer and application. By enabling optional validation layers and using the `VK_EXT_debug_report` extension, developers can obtain much more detailed feedback on the application’s use of Vulkan. This extension defines a way for layers and the implementation to call back to the application for events of interest to the application.

New Object Types

- `VkDebugReportCallbackEXT`

New Commands

- `vkCreateDebugReportCallbackEXT`
- `vkDebugReportMessageEXT`
- `vkDestroyDebugReportCallbackEXT`

New Structures

- Extending `VkInstanceCreateInfo`:
  - `VkDebugReportCallbackCreateInfoEXT`

New Function Pointers

- `PFN_vkDebugReportCallbackEXT`

New Enums

- `VkDebugReportFlagBitsEXT`
- `VkDebugReportObjectTypeEXT`

New Bitmasks

- `VkDebugReportFlagsEXT`

New Enum Constants

- `VK_EXT_DEBUG_REPORT_EXTENSION_NAME`
- `VK_EXT_DEBUG_REPORT_SPEC_VERSION`

Extending `VkObjectType`:

- `VK_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT`

Extending `VkResult`:

- `VK_ERROR_VALIDATION_FAILED_EXT`

Extending `VkStructureType`:
If Version 1.1 is supported:

- Extending VkDebugReportObjectTypeEXT:
  - VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_EXT
  - VK_DEBUG_REPORT_OBJECT_TYPE_SAMPLER_YCBCR_CONVERSION_EXT

Examples

VK_EXT_debug_report allows an application to register multiple callbacks with the validation layers. Some callbacks may log the information to a file, others may cause a debug break point or other application defined behavior. An application can register callbacks even when no validation layers are enabled, but they will only be called for loader and, if implemented, driver events.

To capture events that occur while creating or destroying an instance an application can link a VkDebugReportCallbackCreateInfoEXT structure to the pNext element of the VkInstanceCreateInfo structure given to vkCreateInstance. This callback is only valid for the duration of the vkCreateInstance and the vkDestroyInstance call. Use vkCreateDebugReportCallbackEXT to create persistent callback objects.

Example uses: Create three callback objects. One will log errors and warnings to the debug console using Windows OutputDebugString. The second will cause the debugger to break at that callback when an error happens and the third will log warnings to stdout.
VkResult res;
VkDebugReportCallbackEXT cb1, cb2, cb3;

VkDebugReportCallbackCreateInfoEXT callback1 = {
    VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT,   // sType
    NULL,                                                        // pNext
    VK_DEBUG_REPORT_ERROR_BIT_EXT | VK_DEBUG_REPORT_WARNING_BIT_EXT,   // flags
    myOutputDebugString,                                         // pfnCallback
    NULL                                                         // pUserData
};
res = vkCreateDebugReportCallbackEXT(instance, &callback1, &cb1);
if (res != VK_SUCCESS)
    /* Do error handling for VK_ERROR_OUT_OF_MEMORY */

    callback.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT;
callback.pfnCallback = myDebugBreak;
callback.pUserData = NULL;
res = vkCreateDebugReportCallbackEXT(instance, &callback, &cb2);
if (res != VK_SUCCESS)
    /* Do error handling for VK_ERROR_OUT_OF_MEMORY */

VkDebugReportCallbackCreateInfoEXT callback3 = {
    VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT,   // sType
    NULL,                                                        // pNext
    VK_DEBUG_REPORT_WARNING_BIT_EXT,                            // flags
    mystdOutLogger,                                             // pfnCallback
    NULL                                                         // pUserData
};
res = vkCreateDebugReportCallbackEXT(instance, &callback3, &cb3);
if (res != VK_SUCCESS)
    /* Do error handling for VK_ERROR_OUT_OF_MEMORY */

...}

/* remove callbacks when cleaning up */
vkDestroyDebugReportCallbackEXT(instance, cb1);
vkDestroyDebugReportCallbackEXT(instance, cb2);
vkDestroyDebugReportCallbackEXT(instance, cb3);

Note
In the initial release of the VK_EXT_debug_report extension, the token
VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT was used. Starting in version 2 of
the extension branch, VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT is
used instead for consistency with Vulkan naming rules. The older enum is still
available for backwards compatibility.
Note

In the initial release of the VK_EXT_debug_report extension, the token VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_EXT was used. Starting in version 8 of the extension branch, VK_DEBUG_REPORT_OBJECT_TYPE_DEBUG_REPORT_CALLBACK_EXT_EXT is used instead for consistency with Vulkan naming rules. The older enum is still available for backwards compatibility.

Issues

1) What is the hierarchy / seriousness of the message flags? E.g. ERROR > WARN > PERF_WARN ...

**RESOLVED:** There is no specific hierarchy. Each bit is independent and should be checked via bitwise AND. For example:

```c
if (localFlags & VK_DEBUG_REPORT_ERROR_BIT_EXT) {
    process error message
}
if (localFlags & VK_DEBUG_REPORT_DEBUG_BIT_EXT) {
    process debug message
}
```

The validation layers do use them in a hierarchical way (ERROR > WARN > PERF, WARN > DEBUG > INFO) and they (at least at the time of this writing) only set one bit at a time. But it is not a requirement of this extension.

It is possible that a layer may intercept and change, or augment the flags with extension values the application’s debug report handler may not be familiar with, so it is important to treat each flag independently.

2) Should there be a VU requiring VkDebugReportCallbackCreateInfoEXT::flags to be non-zero?

**RESOLVED:** It may not be very useful, but we do not need VU statement requiring the VkDebugReportCallbackCreateInfoEXT::msgFlags at create-time to be non-zero. One can imagine that apps may prefer it as it allows them to set the mask as desired - including nothing - at runtime without having to check.

3) What is the difference between VK_DEBUG_REPORT_DEBUG_BIT_EXT and VK_DEBUG_REPORT_INFORMATION_BIT_EXT?

**RESOLVED:** VK_DEBUG_REPORT_DEBUG_BIT_EXT specifies information that could be useful debugging the Vulkan implementation itself.

4) How do you compare handles returned by the debug_report callback to the application’s handles?

**RESOLVED:** Due to the different nature of dispatchable and nondispatchable handles there is no generic way (that we know of) that works for common compilers with 32bit, 64bit, C and C++. We recommend applications use the same cast that the validation layers use:
reinterpret_cast<uint64_t &>(dispatchableHandle)
(uint64_t)(nondispatchableHandle)

+ This does require that the app treat dispatchable and nondispatchable handles differently.

**Version History**

- Revision 1, 2015-05-20 (Courtney Goetzenleuchter)
  - Initial draft, based on LunarG KHR spec, other KHR specs
- Revision 2, 2016-02-16 (Courtney Goetzenleuchter)
  - Update usage, documentation
- Revision 3, 2016-06-14 (Courtney Goetzenleuchter)
  - Update VK_EXT_DEBUG_REPORT_SPEC_VERSION to indicate added support for vkCreateInstance and vkDestroyInstance
- Revision 4, 2016-12-08 (Mark Lobodzinski)
  - Added Display_KHR, DisplayModeKHR extension objects
  - Added ObjectTable_NVX, IndirectCommandsLayout_NVX extension objects
  - Bumped spec revision
  - Retroactively added version history
- Revision 5, 2017-01-31 (Baldur Karlsson)
  - Moved definition of VkDebugReportObjectTypeEXT from debug marker chapter
- Revision 6, 2017-01-31 (Baldur Karlsson)
  - Added VK_DEBUG_REPORT_OBJECT_TYPE_DESCRIPTOR_UPDATE_TEMPLATE_KHR_EXT
- Revision 7, 2017-04-20 (Courtney Goeltzenleuchter)
  - Clarify wording and address questions from developers.
- Revision 8, 2017-04-21 (Courtney Goeltzenleuchter)
  - Remove unused enum VkDebugReportErrorEXT
- Revision 9, 2017-09-12 (Tobias Hector)
  - Added interactions with Vulkan 1.1
- Revision 10, 2020-12-14 (Courtney Goetzenleuchter)
  - Add issue 4 discussing matching handles returned by the extension, based on suggestion in public issue 368.

**VK_EXT_shader_subgroup_vote**
Name String
VK_EXT_shader_subgroup_vote

Extension Type
Device extension

Registered Extension Number
66

Revision
1

Extension and Version Dependencies
• Requires Vulkan 1.0

Deprecation state
• Deprecated by Vulkan 1.1

Contact
• Daniel Koch @dgkoch

Other Extension Metadata

Last Modified Date
2016-11-28

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_KHR_subgroup_vote
• This extension provides API support for GL_ARB_shader_group_vote

Contributors
• Neil Henning, Codeplay
• Daniel Koch, NVIDIA Corporation

Description

This extension adds support for the following SPIR-V extension in Vulkan:

• SPV_KHR_subgroup_vote

This extension provides new SPIR-V instructions:

• OpSubgroupAllKHR,
• OpSubgroupAnyKHR, and
• **OpSubgroupAllEqualKHR.**

to compute the composite of a set of boolean conditions across a group of shader invocations that are running concurrently (a *subgroup*). These composite results may be used to execute shaders more efficiently on a *VkPhysicalDevice*.

When using GLSL source-based shader languages, the following shader functions from GL_ARB_shader_group_vote can map to these SPIR-V instructions:

• **anyInvocationARB() → OpSubgroupAnyKHR,**
• **allInvocationsARB() → OpSubgroupAllKHR,** and
• **allInvocationsEqualARB() → OpSubgroupAllEqualKHR.**

The subgroup across which the boolean conditions are evaluated is implementation-dependent, and this extension provides no guarantee over how individual shader invocations are assigned to subgroups. In particular, a subgroup has no necessary relationship with the compute shader *local workgroup* — any pair of shader invocations in a compute local workgroup may execute in different subgroups as used by these instructions.

Compute shaders operate on an explicitly specified group of threads (a local workgroup), but many implementations will also group non-compute shader invocations and execute them concurrently. When executing code like

```plaintext
if (condition) {
    result = do_fast_path();
} else {
    result = do_general_path();
}
```

where *condition* diverges between invocations, an implementation might first execute *do_fast_path()* for the invocations where *condition* is true and leave the other invocations dormant. Once *do_fast_path()* returns, it might call *do_general_path()* for invocations where *condition* is *false* and leave the other invocations dormant. In this case, the shader executes both the fast and the general path and might be better off just using the general path for all invocations.

This extension provides the ability to avoid divergent execution by evaluating a condition across an entire subgroup using code like:

```plaintext
if (allInvocationsARB(condition)) {
    result = do_fast_path();
} else {
    result = do_general_path();
}
```

The built-in function *allInvocationsARB()* will return the same value for all invocations in the group, so the group will either execute *do_fast_path()* or *do_general_path()*, but never both. For example, shader code might want to evaluate a complex function iteratively by starting with an
approximation of the result and then refining the approximation. Some input values may require a small number of iterations to generate an accurate result (do_fast_path) while others require a larger number (do_general_path). In another example, shader code might want to evaluate a complex function (do_general_path) that can be greatly simplified when assuming a specific value for one of its inputs (do_fast_path).

**Deprecated by Vulkan 1.1**

All functionality in this extension is superseded by the core Vulkan 1.1 subgroup operations.

**New Enum Constants**

- VK_EXT_SHADER_SUBGROUP_VOTE_EXTENSION_NAME
- VK_EXT_SHADER_SUBGROUP_VOTE_SPEC_VERSION

**New SPIR-V Capabilities**

- SubgroupVoteKHR

**Version History**

- Revision 1, 2016-11-28 (Daniel Koch)
  - Initial draft

**VK_EXT_validation_flags**

**Name String**

VK_EXT_validation_flags

**Extension Type**

Instance extension

**Registered Extension Number**

62

**Revision**

2

**Extension and Version Dependencies**

- Requires Vulkan 1.0

**Deprecation state**

- Deprecated by VK_EXT_validation_features extension

**Special Use**

- Debugging tools
Description

This extension provides the `VkValidationFlagsEXT` struct that can be included in the `pNext` chain of the `VkInstanceCreateInfo` structure passed as the `pCreateInfo` parameter of `vkCreateInstance`. The structure contains an array of `VkValidationCheckEXT` values that will be disabled by the validation layers.

Deprecation by `VK_EXT_validation_features`

Functionality in this extension is subsumed into the `VK_EXT_validation_features` extension.

New Structures

- Extending `VkInstanceCreateInfo`:
  - `VkValidationFlagsEXT`

New Enums

- `VkValidationCheckEXT`

New Enum Constants

- `VK_EXT_VALIDATION_FLAGS_EXTENSION_NAME`
- `VK_EXT_VALIDATION_FLAGS_SPEC_VERSION`

Extending `VkStructureType`:

- `VK_STRUCTURE_TYPE_VALIDATION_FLAGS_EXT`

Version History

- Revision 2, 2019-08-19 (Mark Lobodzinski)
  - Marked as deprecated
VK_AMD_draw_indirect_count

Name String
VK_AMD_draw_indirect_count

Extension Type
Device extension

Registered Extension Number
34

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0

Deprecation state
• Promoted to VK_KHR_draw_indirect_count extension
  • Which in turn was promoted to Vulkan 1.2

Contact
• Daniel Rakos drakos-amd

Other Extension Metadata

Last Modified Date
2016-08-23

Interactions and External Dependencies
• Promoted to VK_KHR_draw_indirect_count

IP Status
No known IP claims.

Contributors
• Matthaeus G. Chajdas, AMD
• Derrick Owens, AMD
• Graham Sellers, AMD
• Daniel Rakos, AMD
• Dominik Witczak, AMD
Description

This extension allows an application to source the number of draw calls for indirect draw calls from a buffer. This enables applications to generate arbitrary amounts of draw commands and execute them without host intervention.

Promotion to VK_KHR_draw_indirect_count

All functionality in this extension is included in VK_KHR_draw_indirect_count, with the suffix changed to KHR. The original type, enum and command names are still available as aliases of the core functionality.

New Commands

- `vkCmdDrawIndexedIndirectCountAMD`
- `vkCmdDrawIndirectCountAMD`

New Enum Constants

- `VK_AMD_DRAW_INDIRECT_COUNT_EXTENSION_NAME`
- `VK_AMD_DRAW_INDIRECT_COUNT_SPEC_VERSION`

Version History

- Revision 2, 2016-08-23 (Dominik Witczak)
  - Minor fixes
- Revision 1, 2016-07-21 (Matthaeus Chajdas)
  - Initial draft

VK_AMD_gpu_shader_half_float

Name String

`VK_AMD_gpu_shader_half_float`

Extension Type

Device extension

Registered Extension Number

37

Revision

2

Extension and Version Dependencies

- Requires Vulkan 1.0
Deprecation state

- Deprecated by VK_KHR_shader_float16_int8 extension
  - Which in turn was promoted to Vulkan 1.2

Contact

- Dominik Witczak dominikwitczakamd

Other Extension Metadata

Last Modified Date

- 2019-04-11

IP Status

- No known IP claims.

Interactions and External Dependencies

- This extension requires SPV_AMD_gpu_shader_half_float

Contributors

- Daniel Rakos, AMD
- Dominik Witczak, AMD
- Donglin Wei, AMD
- Graham Sellers, AMD
- Qun Lin, AMD
- Rex Xu, AMD

Description

This extension adds support for using half float variables in shaders.

Deprecation by VK_KHR_shader_float16_int8

Functionality in this extension was included in VK_KHR_shader_float16_int8 extension, when VkPhysicalDeviceShaderFloat16Int8FeaturesKHR::shaderFloat16 is enabled.

New Enum Constants

- VK_AMD_GPU_SHADER_HALF_FLOAT_EXTENSION_NAME
- VK_AMD_GPU_SHADER_HALF_FLOAT_SPEC_VERSION

Version History

- Revision 2, 2019-04-11 (Tobias Hector)
  - Marked as deprecated
- Revision 1, 2016-09-21 (Dominik Witczak)
VK_AMD_gpu_shader_int16

Name String
VK_AMD_gpu_shader_int16

Extension Type
Device extension

Registered Extension Number
133

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0

Deprecation state
• Deprecated by VK_KHR_shader_float16_int8 extension
  • Which in turn was promoted to Vulkan 1.2

Contact
• Qun Lin @linqun

Other Extension Metadata

Last Modified Date
2019-04-11

IP Status
No known IP claims.

Interactions and External Dependencies
• This extension requires SPV_AMD_gpu_shader_int16

Contributors
• Daniel Rakos, AMD
• Dominik Witczak, AMD
• Matthaeus G. Chajdas, AMD
• Rex Xu, AMD
• Timothy Lottes, AMD
• Zhi Cai, AMD
External Dependencies

- SPV_AMD_gpu_shader_int16

Description

This extension adds support for using 16-bit integer variables in shaders.

Deprecation by VK_KHR_shader_float16_int8

Functionality in this extension was included in VK_KHR_shader_float16_int8 extension, when VkPhysicalDeviceFeatures::shaderInt16 and VkPhysicalDeviceShaderFloat16Int8FeaturesKHR::shaderFloat16 are enabled.

New Enum Constants

- VK_AMD_GPU_SHADER_INT16_EXTENSION_NAME
- VK_AMD_GPU_SHADER_INT16_SPEC_VERSION

Version History

- Revision 2, 2019-04-11 (Tobias Hector)
  - Marked as deprecated
- Revision 1, 2017-06-18 (Dominik Witczak)
  - First version

VK_AMD_negative_viewport_height

Name String

VK_AMD_negative_viewport_height

Extension Type

Device extension

Registered Extension Number

36

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Obsoleted by VK_KHR_maintenance1 extension
  - Which in turn was promoted to Vulkan 1.1
Description

This extension allows an application to specify a negative viewport height. The result is that the viewport transformation will flip along the y-axis.

- Allow negative height to be specified in the VkViewport::height field to perform y-inversion of the clip-space to framebuffer-space transform. This allows apps to avoid having to use gl_Position.y = -gl_Position.y in shaders also targeting other APIs.

Obsoletion by VK_KHR_maintenance1 and Vulkan 1.1

Functionality in this extension is included in VK_KHR_maintenance1 and subsequently Vulkan 1.1. Due to some slight behavioral differences, this extension must not be enabled alongside VK_KHR_maintenance1, or in an instance created with version 1.1 or later requested in VkApplicationInfo::apiVersion.

New Enum Constants

- VK_AMD_NEGATIVE_VIEWPORT_HEIGHT_EXTENSION_NAME
- VK_AMD_NEGATIVE_VIEWPORT_HEIGHT_SPEC_VERSION

Version History

- Revision 1, 2016-09-02 (Matthaeus Chajdas)
  - Initial draft

VK_MVK_ios_surface

Name String

VK_MVK_ios_surface
The **VK_MVK_ios_surface** extension is an instance extension. It provides a mechanism to create a `VkSurfaceKHR` object (defined by the **VK_KHR_surface** extension) based on a `UIView`, the native surface type of iOS, which is underpinned by a `CAMetalLayer`, to support rendering to the surface using Apple's Metal framework.

**Deprecation by **VK_EXT_metal_surface** extension

The **VK_MVK_ios_surface** extension is considered deprecated and has been superseded by the **VK_EXT_metal_surface** extension.

**New Commands**

- `vkCreateIOSSurfaceMVK`
New Structures

- VkIOSSurfaceCreateInfoMVK

New Bitmasks

- VkIOSSurfaceCreateFlagsMVK

New Enum Constants

- VK_MVK_IOS_SURFACE_EXTENSION_NAME
- VK_MVK_IOS_SURFACE_SPEC_VERSION
- Extending VkStructureType:
  - VK_STRUCTURE_TYPE_IOS_SURFACE_CREATE_INFO_MVK

Version History

- Revision 1, 2017-02-15 (Bill Hollings)
  - Initial draft.
- Revision 2, 2017-02-24 (Bill Hollings)
  - Minor syntax fix to emphasize firm requirement for UIView to be backed by a CAMetalLayer.
- Revision 3, 2020-07-31 (Bill Hollings)
  - Update documentation on requirements for UIView.
  - Mark as deprecated by VK_EXT_metal_surface.

VK_MVK_macos_surface

Name String

VK_MVK_macos_surface

Extension Type

Instance extension

Registered Extension Number

124

Revision

3

Extension and Version Dependencies

- Requires Vulkan 1.0
- Requires VK_KHR_surface
Deprecation state

• Deprecated by VK_EXT_metal_surface extension

Contact

• Bill Hollings billhollings

Other Extension Metadata

Last Modified Date

2020-07-31

IP Status

No known IP claims.

Contributors

• Bill Hollings, The Brenwill Workshop Ltd.

Description

The VK_MVK_macos_surface extension is an instance extension. It provides a mechanism to create a VkSurfaceKHR object (defined by the VK_KHR_surface extension) based on an NSView, the native surface type of macOS, which is underpinned by a CAMetalLayer, to support rendering to the surface using Apple’s Metal framework.

Deprecation by VK_EXT_metal_surface

The VK_MVK_macos_surface extension is considered deprecated and has been superseded by the VK_EXT_metal_surface extension.

New Commands

• vkCreateMacOSSurfaceMVK

New Structures

• VkMacOSSurfaceCreateInfoMVK

New Bitmasks

• VkMacOSSurfaceCreateFlagsMVK

New Enum Constants

• VK_MVK_MACOS_SURFACE_EXTENSION_NAME
• VK_MVK_MACOS_SURFACE_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK
Version History

- Revision 1, 2017-02-15 (Bill Hollings)
  - Initial draft.
- Revision 2, 2017-02-24 (Bill Hollings)
  - Minor syntax fix to emphasize firm requirement for `NSView` to be backed by a `CAMetalLayer`.
- Revision 3, 2020-07-31 (Bill Hollings)
  - Update documentation on requirements for `NSView`.
  - Mark as deprecated by `VK_EXT_metal_surface`.

VK_NV_dedicated_allocation

Name String

`VK_NV_dedicated_allocation`

Extension Type

Device extension

Registered Extension Number

27

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- *Deprecated* by `VK_KHR_dedicated_allocation` extension
  - Which in turn was *promoted* to *Vulkan 1.1*

Contact

- Jeff Bolz [jeffbolznv](mailto:jeffbolznv)

Other Extension Metadata

Last Modified Date

2016-05-31

IP Status

No known IP claims.

Contributors

- Jeff Bolz, NVIDIA
Description

This extension allows device memory to be allocated for a particular buffer or image resource, which on some devices can significantly improve the performance of that resource. Normal device memory allocations must support memory aliasing and sparse binding, which could interfere with optimizations like framebuffer compression or efficient page table usage. This is important for render targets and very large resources, but need not (and probably should not) be used for smaller resources that can benefit from suballocation.

This extension adds a few small structures to resource creation and memory allocation: a new structure that flags whether an image/buffer will have a dedicated allocation, and a structure indicating the image or buffer that an allocation will be bound to.

New Structures

- Extending `VkBufferCreateInfo`:
  - `VkDedicatedAllocationBufferCreateInfoNV`

- Extending `VkImageCreateInfo`:
  - `VkDedicatedAllocationImageCreateInfoNV`

- Extending `VkMemoryAllocateInfo`:
  - `VkDedicatedAllocationMemoryAllocateInfoNV`

New Enum Constants

- `VK_NV_DEDICATED_ALLOCATION_EXTENSION_NAME`
- `VK_NV_DEDICATED_ALLOCATION_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_BUFFER_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV`

Examples

```c
// Create an image with
// VkDedicatedAllocationImageCreateInfoNV::dedicatedAllocation
// set to VK_TRUE

VkDedicatedAllocationImageCreateInfoNV dedicatedImageInfo = {
    VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV,  // sType
    NULL,  // pNext
    VK_TRUE,  // dedicatedAllocation
};
```
VkImageCreateInfo imageCreateInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO, // sType
    &dedicatedImageInfo,                // pNext
    // Other members set as usual
};

VkImage image;
VkResult result = vkCreateImage(
    device,
    &imageCreateInfo,
    NULL,                                  // pAllocator
    &image);

VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(
    device,
    image,
    &memoryRequirements);

// Allocate memory with VkDedicatedAllocationMemoryAllocateInfoNV::image
// pointing to the image we are allocating the memory for

VkDedicatedAllocationMemoryAllocateInfoNV dedicatedInfo =
{
    VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV, // sType
    NULL,                                                               // pNext
    image,                                                              // image
    VK_NULL_HANDLE,                                                      // buffer
};

VkMemoryAllocateInfo memoryAllocateInfo =
{
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO, // sType
    &dedicatedInfo,                         // pNext
    memoryRequirements.size,               // allocationSize
    FindMemoryTypeIndex(memoryRequirements.memoryTypeBits), // memoryTypeIndex
};

VkDeviceMemory memory;
vkAllocateMemory(
    device,
    &memoryAllocateInfo,
    NULL,                                  // pAllocator
    &memory);
// Bind the image to the memory

vkBindImageMemory(
    device,
    image,
    memory,
    0);

Version History

• Revision 1, 2016-05-31 (Jeff Bolz)
  ◦ Internal revisions

VK_NV_external_memory

Name String

VK_NV_external_memory

Extension Type

Device extension

Registered Extension Number

57

Revision

1

Extension and Version Dependencies

• Requires Vulkan 1.0
• Requires VK_NV_external_memory_capabilities

Deprecation state

• Deprecated by VK_KHR_external_memory extension
  ◦ Which in turn was promoted to Vulkan 1.1

Contact

• James Jones cubanismo

Other Extension Metadata

Last Modified Date

2016-08-19

IP Status

No known IP claims.
Contributors

- James Jones, NVIDIA
- Carsten Rohde, NVIDIA

Description

Applications may wish to export memory to other Vulkan instances or other APIs, or import memory from other Vulkan instances or other APIs to enable Vulkan workloads to be split up across application module, process, or API boundaries. This extension enables applications to create exportable Vulkan memory objects such that the underlying resources can be referenced outside the Vulkan instance that created them.

New Structures

- Extending `VkImageCreateInfo`:
  - `VkExternalMemoryImageCreateInfoNV`
- Extending `VkMemoryAllocateInfo`:
  - `VkExportMemoryAllocateInfoNV`

New Enum Constants

- `VK_NV_EXTERNAL_MEMORY_EXTENSION_NAME`
- `VK_NV_EXTERNAL_MEMORY_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV`
  - `VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV`

Issues

1) If memory objects are shared between processes and APIs, is this considered aliasing according to the rules outlined in the Memory Aliasing section?

**RESOLVED**: Yes, but strict exceptions to the rules are added to allow some forms of aliasing in these cases. Further, other extensions may build upon these new aliasing rules to define specific support usage within Vulkan for imported native memory objects, or memory objects from other APIs.

2) Are new image layouts or metadata required to specify image layouts and layout transitions compatible with non-Vulkan APIs, or with other instances of the same Vulkan driver?

**RESOLVED**: No. Separate instances of the same Vulkan driver running on the same GPU should have identical internal layout semantics, so applications have the tools they need to ensure views of images are consistent between the two instances. Other APIs will fall into two categories: Those that are Vulkan compatible (a term to be defined by subsequent interoperability extensions), or Vulkan incompatible. When sharing images with Vulkan incompatible APIs, the Vulkan image must be transitioned to the `VK_IMAGE_LAYOUT_GENERAL` layout before handing it off to the external API.
Note this does not attempt to address cross-device transitions, nor transitions to engines on the same device which are not visible within the Vulkan API. Both of these are beyond the scope of this extension.

Examples

```
// TODO: Write some sample code here.
```

Version History

- Revision 1, 2016-08-19 (James Jones)
  - Initial draft

VK_NV_external_memory_capabilities

Name String

VK_NV_external_memory_capabilities

Extension Type

Instance extension

Registered Extension Number

56

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Deprecated by VK_KHR_external_memory_capabilities extension
  - Which in turn was promoted to Vulkan 1.1

Contact

- James Jones @cubanismo

Other Extension Metadata

Last Modified Date

2016-08-19

IP Status

No known IP claims.
Interactions and External Dependencies

- Interacts with Vulkan 1.1.
- Interacts with `VK_KHR_dedicated_allocation`.
- Interacts with `VK_NV_dedicated_allocation`.

Contributors

- James Jones, NVIDIA

Description

Applications may wish to import memory from the Direct 3D API, or export memory to other Vulkan instances. This extension provides a set of capability queries that allow applications to determine what types of win32 memory handles an implementation supports for a given set of use cases.

New Commands

- `vkGetPhysicalDeviceExternalImageFormatPropertiesNV`

New Structures

- `VkExternalImageFormatPropertiesNV`

New Enums

- `VkExternalMemoryFeatureFlagBitsNV`
- `VkExternalMemoryHandleTypeFlagBitsNV`

New Bitmasks

- `VkExternalMemoryFeatureFlagsNV`
- `VkExternalMemoryHandleTypeFlagsNV`

New Enum Constants

- `VK_NV_EXTERNAL_MEMORY_CAPABILITIES_EXTENSION_NAME`
- `VK_NV_EXTERNAL_MEMORY_CAPABILITIES_SPEC_VERSION`

Issues

1) Why do so many external memory capabilities need to be queried on a per-memory-handle-type basis?

**RESOLVED:** This is because some handle types are based on OS-native objects that have far more limited capabilities than the very generic Vulkan memory objects. Not all memory handle types can name memory objects that support 3D images, for example. Some handle types cannot even support the deferred image and memory binding behavior of Vulkan and require specifying the
image when allocating or importing the memory object.

2) Does the `VkExternalImageFormatPropertiesNV` struct need to include a list of memory type bits that support the given handle type?

**RESOLVED:** No. The memory types that do not support the handle types will simply be filtered out of the results returned by `vkGetImageMemoryRequirements` when a set of handle types was specified at image creation time.

3) Should the non-opaque handle types be moved to their own extension?

**RESOLVED:** Perhaps. However, defining the handle type bits does very little and does not require any platform-specific types on its own, and it is easier to maintain the bitmask values in a single extension for now. Presumably more handle types could be added by separate extensions though, and it would be midly weird to have some platform-specific ones defined in the core spec and some in extensions

**Version History**

- Revision 1, 2016-08-19 (James Jones)
  - Initial version

**VK_NV_external_memory_win32**

**Name String**

`VK_NV_external_memory_win32`

**Extension Type**

Device extension

**Registered Extension Number**

58

**Revision**

1

**Extension and Version Dependencies**

- Requires Vulkan 1.0
- Requires `VK_NV_external_memory`

**Deprecation state**

- Deprecated by `VK_KHR_external_memory_win32` extension

**Contact**

- James Jones @cubanismo
Other Extension Metadata

Last Modified Date
2016-08-19

IP Status
No known IP claims.

Contributors
• James Jones, NVIDIA
• Carsten Rohde, NVIDIA

Description
Applications may wish to export memory to other Vulkan instances or other APIs, or import memory from other Vulkan instances or other APIs to enable Vulkan workloads to be split up across application module, process, or API boundaries. This extension enables win32 applications to export win32 handles from Vulkan memory objects such that the underlying resources can be referenced outside the Vulkan instance that created them, and import win32 handles created in the Direct3D API to Vulkan memory objects.

New Commands
• vkGetMemoryWin32HandleNV

New Structures
• Extending VkMemoryAllocateInfo:
  ◦ VkExportMemoryWin32HandleInfoNV
  ◦ VkImportMemoryWin32HandleInfoNV

New Enum Constants
• VK_NV_EXTERNAL_MEMORY_WIN32_EXTENSION_NAME
• VK_NV_EXTERNAL_MEMORY_WIN32_SPEC_VERSION
• Extending VkStructureType:
  ◦ VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_NV
  ◦ VK_STRUCTURE_TYPE_IMPORT_MEMORY_WIN32_HANDLE_INFO_NV

Issues
1) If memory objects are shared between processes and APIs, is this considered aliasing according to the rules outlined in the Memory Aliasing section?

RESOLVED: Yes, but strict exceptions to the rules are added to allow some forms of aliasing in these cases. Further, other extensions may build upon these new aliasing rules to define specific support
usage within Vulkan for imported native memory objects, or memory objects from other APIs.

2) Are new image layouts or metadata required to specify image layouts and layout transitions compatible with non-Vulkan APIs, or with other instances of the same Vulkan driver?

**RESOLVED:** No. Separate instances of the same Vulkan driver running on the same GPU should have identical internal layout semantics, so applications have the tools they need to ensure views of images are consistent between the two instances. Other APIs will fall into two categories: Those that are Vulkan compatible (a term to be defined by subsequent interoperability extensions), or Vulkan incompatible. When sharing images with Vulkan incompatible APIs, the Vulkan image must be transitioned to the `VK_IMAGE_LAYOUT_GENERAL` layout before handing it off to the external API.

Note this does not attempt to address cross-device transitions, nor transitions to engines on the same device which are not visible within the Vulkan API. Both of these are beyond the scope of this extension.

3) Do applications need to call `CloseHandle()` on the values returned from `vkGetMemoryWin32HandleNV` when `handleType` is `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV`?

**RESOLVED:** Yes, unless it is passed back in to another driver instance to import the object. A successful get call transfers ownership of the handle to the application, while an import transfers ownership to the associated driver. Destroying the memory object will not destroy the handle or the handle’s reference to the underlying memory resource.

**Examples**

```c
// Create an exportable memory object and export an external
// handle from it.

// Pick an external format and handle type.
static const VkFormat format = VK_FORMAT_R8G8B8A8_UNORM;
static const VkExternalMemoryHandleTypeFlagsNV handleType = VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_NV;

extern VkPhysicalDevice physicalDevice;
extern VkDevice device;

VkPhysicalDeviceMemoryProperties memoryProperties;
VkExternalImageFormatPropertiesNV properties;
VkExternalMemoryImageCreateInfoNV externalMemoryImageCreateInfo;
VkDedicatedAllocationImageCreateInfoNV dedicatedImageCreateInfo;
VkImageCreateInfo imageCreateInfo;
VkImage image;
VkMemoryRequirements imageMemoryRequirements;
uint32_t numMemoryTypes;
uint32_t memoryType;
VkExportMemoryAllocateInfoNV exportMemoryAllocateInfo;
```
VkDedicatedAllocationMemoryAllocateInfoNV dedicatedAllocationInfo;
VkMemoryAllocateInfo memoryAllocateInfo;
VkDeviceMemory memory;
VkResult result;
HANDLE memoryHnd;

// Figure out how many memory types the device supports
vkGetPhysicalDeviceMemoryProperties(physicalDevice,
    &memoryProperties);
numMemoryTypes = memoryProperties.memoryTypeCount;

// Check the external handle type capabilities for the chosen format
// Exportable 2D image support with at least 1 mip level, 1 array
// layer, and VK_SAMPLE_COUNT_1_BIT using optimal tiling and supporting
// texturing and color rendering is required.
result = vkGetPhysicalDeviceExternalImageFormatPropertiesNV(
    physicalDevice,
    format,
    VK_IMAGE_TYPE_2D,
    VK_IMAGE_TILING_OPTIMAL,
    VK_IMAGE_USAGE_SAMPLED_BIT |
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,
    0,
    handleType,
    0);

if ((result != VK_SUCCESS) ||
    !(properties.externalMemoryFeatures &
        VK_EXTERNAL_MEMORY_FEATURE_EXPORTABLE_BIT_NV)) {
    abort();
}

// Set up the external memory image creation info
memset(&externalMemoryImageCreateInfo,
    0, sizeof(externalMemoryImageCreateInfo));
externalMemoryImageCreateInfo.sType =
    VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV;
externalMemoryImageCreateInfo.handleTypes = handleType;
if (properties.externalMemoryFeatures &
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV) {
    memset(&dedicatedImageCreateInfo, 0, sizeof(dedicatedImageCreateInfo));
dedicatedImageCreateInfo.sType =
    VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_IMAGE_CREATE_INFO_NV;
dedicatedImageCreateInfo.dedicatedAllocation = VK_TRUE;
    externalMemoryImageCreateInfo.pNext = &dedicatedImageCreateInfo;
}

// Set up the core image creation info
memset(&imageCreateInfo, 0, sizeof(imageCreateInfo));
imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageCreateInfo.pNext = &externalMemoryImageCreateInfo;
imageCreateInfo.format = format;
imageCreateInfo.extent.width = 64;
imageCreateInfo.extent.height = 64;
imageCreateInfo.extent.depth = 1;
imageCreateInfo.mipLevels = 1;
imageCreateInfo.arrayLayers = 1;
imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageCreateInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
imageCreateInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT | 
                        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
imageCreateInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

vkCreateImage(device, &imageCreateInfo, NULL, &image);

vkGetImageMemoryRequirements(device,
                              image,
                              &imageMemoryRequirements);

// For simplicity, just pick the first compatible memory type.
for (memoryType = 0; memoryType < numMemoryTypes; memoryType++) {
    if ((1 << memoryType) & imageMemoryRequirements.memoryTypeBits) {
        break;
    }
}

// At least one memory type must be supported given the prior external
// handle capability check.
assert(memoryType < numMemoryTypes);

// Allocate the external memory object.
memset(&exportMemoryAllocateInfo, 0, sizeof(exportMemoryAllocateInfo));
exportMemoryAllocateInfo.sType = VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV;
exportMemoryAllocateInfo.handleTypes = handleType;
if (properties.externalMemoryFeatures & 
    VK_EXTERNAL_MEMORY_FEATURE_DEDICATED_ONLY_BIT_NV) {
    memset(&dedicatedAllocationInfo, 0, sizeof(dedicatedAllocationInfo));
    dedicatedAllocationInfo.sType = 
                                VK_STRUCTURE_TYPE_DEDICATED_ALLOCATION_MEMORY_ALLOCATE_INFO_NV;
    dedicatedAllocationInfo.image = image;
    exportMemoryAllocateInfo.pNext = &dedicatedAllocationInfo;
}
memset(&memoryAllocateInfo, 0, sizeof(memoryAllocateInfo));
memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
memoryAllocateInfo.pNext = &exportMemoryAllocateInfo;
memoryAllocateInfo.allocationSize = imageMemoryRequirements.size;
memoryAllocateInfo.memoryTypeIndex = memoryType;

vkAllocateMemory(device, &memoryAllocateInfo, NULL, &memory);

if (!(properties.externalMemoryFeatures &
VK_EXTERNAL_MEMORY_FEATUREDEDICATEDONLY_BIT_NV)) {
    vkBindImageMemory(device, image, memory, 0);
}

// Get the external memory opaque FD handle
vkGetMemoryWin32HandleNV(device, memory, &memoryHnd);

Version History

- Revision 1, 2016-08-11 (James Jones)
  - Initial draft

VK_NV_glsl_shader

Name String

VK_NV_glsl_shader

Extension Type

Device extension

Registered Extension Number

13

Revision

1

Extension and Version Dependencies

- Requires Vulkan 1.0

Deprecation state

- Deprecated without replacement

Contact

- Piers Daniell pdaniell-nv

Other Extension Metadata

Last Modified Date

2016-02-14

IP Status

No known IP claims.

Contributors

- Piers Daniell, NVIDIA
Description

This extension allows GLSL shaders written to the GL_KHR_vulkan_gls1 extension specification to be used instead of SPIR-V. The implementation will automatically detect whether the shader is SPIR-V or GLSL, and compile it appropriately.

Deprecation

Functionality in this extension is outside of the scope of Vulkan and is better served by a compiler library such as glslang. No new implementations will support this extension, so applications should not use it.

New Enum Constants

- VK_NV_GLSL_SHADER_EXTENSION_NAME
- VK_NV_GLSL_SHADER_SPEC_VERSION
- Extending VkResult:
  - VK_ERROR_INVALID_SHADER_NV

Examples

Example 1

Passing in GLSL code

```c
char const vss[] =
    "#version 450 core
    "layout(location = 0) in vec2 aVertex;
    "layout(location = 1) in vec4 aColor;
    "out vec4 vColor;
    "void main()
    "{\n    "    vColor = aColor;
    "    gl_Position = vec4(aVertex, 0, 1);
    "}\n"
;
VkShaderModuleCreateInfo vertexShaderInfo = {
    VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO
};
vertexShaderInfo.codeSize = sizeof vss;
vertexShaderInfo.pCode = vss;
VkShaderModule vertexShader;
vkCreateShaderModule(device, &vertexShaderInfo, 0, &vertexShader);
```

Version History

- Revision 1, 2016-02-14 (Piers Daniell)
  - Initial draft
VK_NV_win32_keyed_mutex

Name String
VK_NV_win32_keyed_mutex

Extension Type
Device extension

Registered Extension Number
59

Revision
2

Extension and Version Dependencies
• Requires Vulkan 1.0
• Requires VK_NV_external_memory_win32

Deprecation state
• Promoted to VK_KHR_win32_keyed_mutex extension

Contact
• Carsten Rohde crohde

Other Extension Metadata

Last Modified Date
2016-08-19

IP Status
No known IP claims.

Contributors
• James Jones, NVIDIA
• Carsten Rohde, NVIDIA

Description
Applications that wish to import Direct3D 11 memory objects into the Vulkan API may wish to use the native keyed mutex mechanism to synchronize access to the memory between Vulkan and Direct3D. This extension provides a way for an application to access the keyed mutex associated with an imported Vulkan memory object when submitting command buffers to a queue.

New Structures

• Extending VkSubmitInfo, VkSubmitInfo2KHR:
  ◦ VkWin32KeyedMutexAcquireReleaseInfoNV
New Enum Constants

- `VK_NV_WIN32_KEYED_MUTEX_EXTENSION_NAME`
- `VK_NV_WIN32_KEYED_MUTEX_SPEC_VERSION`
- Extending `VkStructureType`:
  - `VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV`

Examples

```c
// Import a memory object from Direct3D 11, and synchronize
// access to it in Vulkan using keyed mutex objects.

extern VkPhysicalDevice physicalDevice;
extern VkDevice device;
extern HANDLE sharedNtHandle;

static const VkFormat format = VK_FORMAT_R8G8B8A8_UNORM;
static const VkExternalMemoryHandleTypeFlagsNV handleType =
  VK_EXTERNAL_MEMORY_HANDLE_TYPE_D3D11_IMAGE_BIT_NV;

VkPhysicalDeviceMemoryProperties memoryProperties;
VkExternalImageFormatPropertiesNV properties;
VkExternalMemoryImageCreateInfoNV externalMemoryImageCreateInfo;
VkImageCreateInfo imageCreateInfo;
VkImage image;
VkMemoryRequirements imageMemoryRequirements;
uint32_t numMemoryTypes;
uint32_t memoryType;
VkImportMemoryWin32HandleInfoNV importMemoryInfo;
VkMemoryAllocateInfo memoryAllocateInfo;
VkDeviceMemory mem;
VkResult result;

// Figure out how many memory types the device supports
vkGetPhysicalDeviceMemoryProperties(physicalDevice,
  &memoryProperties);
numMemoryTypes = memoryProperties.memoryTypeCount;

// Check the external handle type capabilities for the chosen format
// Importable 2D image support with at least 1 mip level, 1 array
// layer, and VK_SAMPLE_COUNT_1_BIT using optimal tiling and supporting
// texturing and color rendering is required.
result = vkGetPhysicalDeviceExternalImageFormatPropertiesNV(
  physicalDevice,
  format,
  VK_IMAGE_TYPE_2D,
  VK_IMAGE_TILING_OPTIMAL,
  numMemoryTypes,
  handleType,
  &properties);
```
if ((result != VK_SUCCESS) ||
  !(properties.externalMemoryFeatures &
    VK_EXTERNAL_MEMORY_FEATURE_IMPORTABLE_BIT_NV)) {
  abort();
}

// Set up the external memory image creation info
memset(&externalMemoryImageCreateInfo,
      0, sizeof(externalMemoryImageCreateInfo));
externalMemoryImageCreateInfo.sType =
  VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMAGE_CREATE_INFO_NV;
externalMemoryImageCreateInfo.handleTypes = handleType;

// Set up the core image creation info
memset(&imageCreateInfo, 0, sizeof(imageCreateInfo));
imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageCreateInfo.pNext = &externalMemoryImageCreateInfo;
imageCreateInfo.format = format;
imageCreateInfo.extent.width = 64;
imageCreateInfo.extent.height = 64;
imageCreateInfo.extent.depth = 1;
imageCreateInfo.mipLevels = 1;
imageCreateInfo.arrayLayers = 1;
imageCreateInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageCreateInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
imageCreateInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT |
  VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
imageCreateInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

vkCreateImage(device, &imageCreateInfo, NULL, &image);
vkGetImageMemoryRequirements(device,
  image, &imageMemoryRequirements);

// For simplicity, just pick the first compatible memory type.
for (memoryType = 0; memoryType < numMemoryTypes; memoryType++) {
  if ((1 << memoryType) & imageMemoryRequirements.memoryTypeBits) {
    break;
  }
}

// At least one memory type must be supported given the prior external
// handle capability check.
assert(memoryType < numMemoryTypes);
// Allocate the external memory object.
memset(&exportMemoryAllocateInfo, 0, sizeof(exportMemoryAllocateInfo));
exportMemoryAllocateInfo.sType = VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_NV;
importMemoryInfo.handleTypes = handleType;
importMemoryInfo.handle = sharedNtHandle;

memset(&memoryAllocateInfo, 0, sizeof(memoryAllocateInfo));
memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
memoryAllocateInfo.pNext = &exportMemoryAllocateInfo;
memoryAllocateInfo.allocationSize = imageMemoryRequirements.size;
memoryAllocateInfo.memoryTypeIndex = memoryType;

vkAllocateMemory(device, &memoryAllocateInfo, NULL, &mem);

vkBindImageMemory(device, image, mem, 0);
...

const uint64_t acquireKey = 1;
const uint32_t timeout = INFINITE;
const uint64_t releaseKey = 2;

VkWin32KeyedMutexAcquireReleaseInfoNV keyedMutex =
    { VK_STRUCTURE_TYPE_WIN32_KEYED_MUTEX_ACQUIRE_RELEASE_INFO_NV };
keyedMutex.acquireCount = 1;
keyedMutex.pAcquireSyncs = &mem;
keyedMutex.pAcquireKeys = &acquireKey;
keyedMutex.pAcquireTimeoutMilliseconds = &timeout;
keyedMutex.releaseCount = 1;
keyedMutex.pReleaseSyncs = &mem;
keyedMutex.pReleaseKeys = &releaseKey;

VkSubmitInfo submit_info = { VK_STRUCTURE_TYPE_SUBMIT_INFO, &keyedMutex };
submit_info.commandBufferCount = 1;
submit_info.pCommandBuffers = &cmd_buf;
vkQueueSubmit(queue, 1, &submit_info, VK_NULL_HANDLE);

Version History

- Revision 2, 2016-08-11 (James Jones)
  - Updated sample code based on the NV external memory extensions.
  - Renamed from NVX to NV extension.
  - Added Overview and Description sections.
  - Updated sample code to use the NV external memory extensions.
- Revision 1, 2016-06-14 (Carsten Rohde)
  - Initial draft.
Appendix F: API Boilerplate

This appendix defines Vulkan API features that are infrastructure required for a complete functional description of Vulkan, but do not logically belong elsewhere in the Specification.

Vulkan Header Files

Vulkan is defined as an API in the C99 language. Khronos provides a corresponding set of header files for applications using the API, which may be used in either C or C++ code. The interface descriptions in the specification are the same as the interfaces defined in these header files, and both are derived from the vk.xml XML API Registry, which is the canonical machine-readable description of the Vulkan API. The Registry, scripts used for processing it into various forms, and documentation of the registry schema are available as described at https://www.khronos.org(registry/vulkan/#apiregistry).

Language bindings for other languages can be defined using the information in the Specification and the Registry. Khronos does not provide any such bindings, but third-party developers have created some additional bindings.

Vulkan Combined API Header vulkan.h (Informative)

Applications normally will include the header vulkan.h. In turn, vulkan.h always includes the following headers:

- vk_platform.h, defining platform-specific macros and headers.
- vulkan_core.h, defining APIs for the Vulkan core and all registered extensions other than window system-specific and provisional extensions, which are included in separate header files.

In addition, specific preprocessor macros defined at the time vulkan.h is included cause header files for the corresponding window system-specific and provisional interfaces to be included, as described below.

Vulkan Platform-Specific Header vk_platform.h (Informative)

Platform-specific macros and interfaces are defined in vk_platform.h. These macros are used to control platform-dependent behavior, and their exact definitions are under the control of specific platforms and Vulkan implementations.

Platform-Specific Calling Conventions

On many platforms the following macros are empty strings, causing platform- and compiler-specific default calling conventions to be used.

VKAPI_ATTR is a macro placed before the return type in Vulkan API function declarations. This macro controls calling conventions for C++11 and GCC/Clang-style compilers.

VKAPI_CALL is a macro placed after the return type in Vulkan API function declarations. This macro controls calling conventions for MSVC-style compilers.
VKAPI_PTR is a macro placed between the '(' and '*' in Vulkan API function pointer declarations. This macro also controls calling conventions, and typically has the same definition as VKAPI_ATTR or VKAPI_CALL, depending on the compiler.

With these macros, a Vulkan function declaration takes the form of:

```c
VKAPI_ATTR <return_type> VKAPI_CALL <command_name>(<command_parameters>);
```

Additionally, a Vulkan function pointer type declaration takes the form of:

```c
typedef <return_type> (VKAPI_PTR *PFN_<command_name>)(<command_parameters>);
```

**Platform-Specific Header Control**

If the VK_NO_STDINT_H macro is defined by the application at compile time, extended integer types used by the Vulkan API, such as `uint8_t`, must also be defined by the application. Otherwise, the Vulkan headers will not compile. If VK_NO_STDINT_H is not defined, the system `<stdint.h>` is used to define these types. There is a fallback path when Microsoft Visual Studio version 2008 and earlier versions are detected at compile time.

If the VK_NO_STDDEF_H macro is defined by the application at compile time, `size_t`, must also be defined by the application. Otherwise, the Vulkan headers will not compile. If VK_NO_STDDEF_H is not defined, the system `<stddef.h>` is used to define this type.

**Vulkan Core API Header vulkan_core.h**

Applications that do not make use of window system-specific extensions may simply include `vulkan_core.h` instead of `vulkan.h`, although there is usually no reason to do so. In addition to the Vulkan API, `vulkan_core.h` also defines a small number of C preprocessor macros that are described below.

**Vulkan Header File Version Number**

`VK_HEADER_VERSION` is the version number of the `vulkan_core.h` header. This value is kept synchronized with the patch version of the released Specification.

```c
// Provided by VK_VERSION_1_0
// Version of this file
#define VK_HEADER_VERSION 177
```

`VK_HEADER_VERSION_COMPLETE` is the complete version number of the `vulkan_core.h` header, comprising the major, minor, and patch versions. The major/minor values are kept synchronized with the complete version of the released Specification. This value is intended for use by automated tools to identify exactly which version of the header was used during their generation.

Applications should not use this value as their `VkApplicationInfo::apiVersion`. Instead applications
should explicitly select a specific fixed major/minor API version using, for example, one of the
\texttt{VK\_API\_VERSION\_*_*} values.

\begin{verbatim}
#define VK\_HEADER\_VERSION\_COMPLETE VK\_MAKE\_API\_VERSION(0, 1, 2, VK\_HEADER\_VERSION)
\end{verbatim}

\texttt{VK\_API\_VERSION} is now commented out of \texttt{vulkan\_core.h} and \textbf{cannot} be used.

\begin{verbatim}
// Provided by VK\_VERSION\_1\_0
// DEPRECATED: This define has been removed. Specific version defines (e.g. VK\_API\_VERSION\_1\_0), or the VK\_MAKE\_VERSION macro, should be used instead.
#define VK\_API\_VERSION VK\_MAKE\_VERSION(1, 0, 0) // Patch version should always be set to 0
\end{verbatim}

\textbf{Vulkan Handle Macros}

\texttt{VK\_DEFINE\_HANDLE} defines a \textit{dispatchable handle} type.

\begin{verbatim}
#define VK\_DEFINE\_HANDLE(object) typedef struct object\_T* object;
\end{verbatim}

- \texttt{object} is the name of the resulting C type.

The only dispatchable handle types are those related to device and instance management, such as \texttt{VkDevice}.

\texttt{VK\_DEFINE\_NON\_DISPATCHABLE\_HANDLE} defines a \textit{non-dispatchable handle} type.

\begin{verbatim}
#ifndef VK\_DEFINE\_NON\_DISPATCHABLE\_HANDLE
#if (VK\_USE\_64\_BIT\_PTR\_DEFINES==1)
#define VK\_DEFINE\_NON\_DISPATCHABLE\_HANDLE(object) typedef struct object\_T
*object;
#else
#define VK\_DEFINE\_NON\_DISPATCHABLE\_HANDLE(object) typedef uint64\_t object;
#endif
#endif
\end{verbatim}

- \texttt{object} is the name of the resulting C type.

Most Vulkan handle types, such as \texttt{VkBuffer}, are non-dispatchable.
The `vulkan_core.h` header allows the `VK_DEFINE_NON_DISPATCHABLE_HANDLE` and `VK_NULL_HANDLE` definitions to be overridden by the application. If `VK_DEFINE_NON_DISPATCHABLE_HANDLE` is already defined when `vulkan_core.h` is compiled, the default definitions for `VK_DEFINE_NON_DISPATCHABLE_HANDLE` and `VK_NULL_HANDLE` are skipped. This allows the application to define a binary-compatible custom handle which may provide more type-safety or other features needed by the application. Applications must not define handles in a way that is not binary compatible - where binary compatibility is platform dependent.

`VK_NULL_HANDLE` is a reserved value representing a non-valid object handle. It may be passed to and returned from Vulkan commands only when specifically allowed.

```c
// Provided by VK_VERSION_1_0

#ifndef VK_DEFINE_NON_DISPATCHABLE_HANDLE
    #if (VK_USE_64_BIT_PTR_DEFINES==1)
        #if __cplusplus >= 201103L || (defined(_MSVC_LANG) && (_MSVC_LANG >= 201103L))
            #define VK_NULL_HANDLE nullptr
        #else
            #define VK_NULL_HANDLE ((void*)0)
        #endif
    #else
        #define VK_NULL_HANDLE NULL
    #endif
#endif
#endif
```

`VK_USE_64_BIT_PTR_DEFINES` defines whether the default non-dispatchable handles are declared using either a 64-bit pointer type or a 64-bit unsigned integer type.

`VK_USE_64_BIT_PTR_DEFINES` is set to '1' to use a 64-bit pointer type or any other value to use a 64-bit unsigned integer type.

```c
// Provided by VK_VERSION_1_0

#ifndef VK_USE_64_BIT_PTR_DEFINES
    #if defined(__LP64__) || defined(_WIN64) || (defined(__x86_64__) && !defined(__ILP32__) ) || defined(_M_X64) || defined(__ia64) || defined (_M_IA64) || defined (__aarch64__) || defined(__powerpc64__)
        #define VK_USE_64_BIT_PTR_DEFINES 1
    #else
        #define VK_USE_64_BIT_PTR_DEFINES 0
    #endif
#endif
```
Note
The `vulkan_core.h` header allows the `VK_USE_64_BIT_PTR_DEFINES` definition to be overridden by the application. This allows the application to select either a 64-bit pointer type or a 64-bit unsigned integer type for non-dispatchable handles in the case where the predefined preprocessor check doesn't identify the desired configuration.

Window System-Specific Header Control (Informative)

To use a Vulkan extension supporting a platform-specific window system, header files for that window systems **must** be included at compile time, or platform-specific types **must** be forward-declared. The Vulkan header files cannot determine whether or not an external header is available at compile time, so platform-specific extensions are provided in separate headers from the core API and platform-independent extensions, allowing applications to decide which ones should be defined and how the external headers are included.

Extensions dependent on particular sets of platform headers, or that forward-declare platform-specific types, are declared in a header named for that platform. Before including these platform-specific Vulkan headers, applications **must** include both `vulkan_core.h` and any external native headers the platform extensions depend on.

As a convenience for applications that do not need the flexibility of separate platform-specific Vulkan headers, `vulkan.h` includes `vulkan_core.h`, and then conditionally includes platform-specific Vulkan headers and the external headers they depend on. Applications control which platform-specific headers are included by defining macros before including `vulkan.h`.

The correspondence between platform-specific extensions, external headers they require, the platform-specific header which declares them, and the preprocessor macros which enable inclusion by `vulkan.h` are shown in the following table.

**Table 94. Window System Extensions and Headers**

<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Window System Name</th>
<th>Platform-specific Header</th>
<th>Required External Headers</th>
<th>Controlling <code>vulkan.h</code> Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_KHR_android_surface</td>
<td>Android</td>
<td><code>vulkan_android.h</code></td>
<td>None</td>
<td><code>VK_USE_PLATFORM_ANDROID_KHR</code></td>
</tr>
<tr>
<td>VK_KHR_wayland_surface</td>
<td>Wayland</td>
<td><code>vulkan_wayland.h</code></td>
<td><code>&lt;wayland-client.h&gt;</code></td>
<td><code>VK_USE_PLATFORM_WAYLAND_KHR</code></td>
</tr>
<tr>
<td>Extension Name</td>
<td>Window System Name</td>
<td>Platform-specific Header</td>
<td>Required External Headers</td>
<td>Controlling vkuse_platform_vulkan.h Macro</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>VK_KHR_win32_surface,</td>
<td>Microsoft Windows</td>
<td>vulkan_win32.h</td>
<td>&lt;windows.h&gt;</td>
<td>VK_USE_PLATFORM_WIN32_KHR</td>
</tr>
<tr>
<td>VK_KHR_external_memory_win32,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_KHR_win32_keyed_mutex,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_KHR_external_semaphore_win32,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_KHR_external_fence_win32,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_NV_external_memory_win32,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_NV_win32_keyed_mutex,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_KHR_xcb_surface,</td>
<td>X11 Xcb</td>
<td>vulkan_xcb.h</td>
<td>&lt;xcb/xcb.h&gt;</td>
<td>VK_USE_PLATFORM_XCB_KHR</td>
</tr>
<tr>
<td>VK_KHR_xlib_surface</td>
<td>X11 Xlib</td>
<td>vulkan_xlib.h</td>
<td>&lt;X11/Xlib.h&gt;</td>
<td>VK_USE_PLATFORM_XLIB_KHR</td>
</tr>
<tr>
<td>VK_EXT_directfb_surface,</td>
<td>DirectFB</td>
<td>vulkan_directfb.h</td>
<td>&lt;directfb/directfb.h&gt;</td>
<td>VK_USE_PLATFORM_DIRECTFB_EXT</td>
</tr>
</tbody>
</table>
| VK_EXT_acquire_xlib_display,           | X11 XRandR         | vulkan_xlib_xrandr.h     | <X11/Xlib.h>, <X11/extensions/Xr
andr.h> | VK_USE_PLATFORM_XLIB_XRAN
DR_EXT |
| VK_GGP_stream_descriptor_surface,      | Google Games       | vulkan_ggp.h             | <ggp_c/vulkan_typ
eses.h>               | VK_USE_PLATFORM_GGP                      |
| VK_GGP_frame_token                     |                    |                          |                          |                                          |
| VK_MVK_ios_surface                     | iOS                | vulkan_ios.h             | None                     | VK_USE_PLATFORM_IOS_MVK                    |
| VK_MVK_macos_surface                   | macOS              | vulkan_macos.h           | None                     | VK_USE_PLATFORM_MACOS_MVK                   |
| VK_NN_vi_surface                       | VI                 | vulkan_vi.h              | None                     | VK_USE_PLATFORM_VI_NN                      |
| VK_FUCHSIA_imagepipe_surface           | Fuchsia            | vulkan_fuchsia.h         | <zircon/types.h>         | VK_USE_PLATFORM_FUCHSIA                    |
| VK_EXT_metal_surface                   | Metal on CoreAnim
ation | vulkan_metal.h          | None                     | VK_USE_PLATFORM_METAL_EXT                  |
| VK_QNX_screen_surface                  | QNX Screen         | vulkan_screen.h          | <screen/screen.h>       | VK_USE_PLATFORM_SCREEN_QNX                  |

**Note**

This section describes the purpose of the headers independently of the specific underlying functionality of the window system extensions themselves. Each extension name will only link to a description of that extension when viewing a specification built with that extension included.
Provisional extensions should not be used in production applications. The functionality defined by such extensions may change in ways that break backwards compatibility between revisions, and before final release of a non-provisional version of that extension.

Provisional extensions are defined in a separate provisional header, VulkanBeta.h, allowing applications to decide whether or not to include them. The mechanism is similar to window system-specific headers: before including VulkanBeta.h, applications must include VulkanCore.h.

Note

Sometimes a provisional extension will include a subset of its interfaces in VulkanCore.h. This may occur if the provisional extension is promoted from an existing vendor or EXT extension and some of the existing interfaces are defined as aliases of the provisional extension interfaces. All other interfaces of that provisional extension which are not aliased will be included in VulkanBeta.h.

As a convenience for applications, Vulkan.h conditionally includes VulkanBeta.h. Applications can control inclusion of VulkanBeta.h by defining the macro VK_ENABLE_BETA_EXTENSIONS before including Vulkan.h.

Note

Starting in version 1.2.171 of the Specification, all provisional enumerants are protected by the macro VK_ENABLE_BETA_EXTENSIONS. Applications needing to use provisional extensions must always define this macro, even if they are explicitly including VulkanBeta.h. This is a minor change to behavior, affecting only provisional extensions.

Note

This section describes the purpose of the provisional header independently of the specific provisional extensions which are contained in that header at any given time. The extension appendices for provisional extensions note their provisional status, and link back to this section for more information. Provisional extensions are intended to provide early access for bleeding-edge developers, with the understanding that extension interfaces may change in response to developer feedback. Provisional extensions are very likely to eventually be updated and released as non-provisional extensions, but there is no guarantee this will happen, or how long it will take if it does happen.
Appendix G: Invariance

The Vulkan specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different Vulkan implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

Repeatability

The obvious and most fundamental case is repeated issuance of a series of Vulkan commands. For any given Vulkan and framebuffer state vector, and for any Vulkan command, the resulting Vulkan and framebuffer state must be identical whenever the command is executed on that initial Vulkan and framebuffer state. This repeatability requirement does not apply when using shaders containing side effects (image and buffer variable stores and atomic operations), because these memory operations are not guaranteed to be processed in a defined order.

The repeatability requirement does not apply for rendering done using a graphics pipeline that uses VK_RASTERIZATION_ORDER_RELAXED_AMD.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different Vulkan mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- “Erasing” a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

Invariance Rules

For a given Vulkan device:

**Rule 1** For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the resulting Vulkan and framebuffer state must be identical each time the command is executed on that initial Vulkan and framebuffer state.
Rule 2 Changes to the following state values have no side effects (the use of any other state value is not affected by the change):

Required:

- Color and depth/stencil attachment contents
- Scissor parameters (other than enable)
- Write masks (color, depth, stencil)
- Clear values (color, depth, stencil)

Strongly suggested:

- Stencil parameters (other than enable)
- Depth test parameters (other than enable)
- Blend parameters (other than enable)
- Logical operation parameters (other than enable)

Corollary 1 Fragment generation is invariant with respect to the state values listed in Rule 2.

Rule 3 The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.

Corollary 2 Images rendered into different color attachments of the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.

Rule 4 Identical pipelines will produce the same result when run multiple times with the same input. The wording “Identical pipelines” means VkPipeline objects that have been created with identical SPIR-V binaries and identical state, which are then used by commands executed using the same Vulkan state vector. Invariance is relaxed for shaders with side effects, such as performing stores or atomics.

Rule 5 All fragment shaders that either conditionally or unconditionally assign FragCoord.z to FragDepth are depth-invariant with respect to each other, for those fragments where the assignment to FragDepth actually is done.

If a sequence of Vulkan commands specifies primitives to be rendered with shaders containing side effects (image and buffer variable stores and atomic operations), invariance rules are relaxed. In particular, rule 1, corollary 2, and rule 4 do not apply in the presence of shader side effects.

The following weaker versions of rules 1 and 4 apply to Vulkan commands involving shader side effects:

Rule 6 For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the contents of any framebuffer state not directly or indirectly affected by results of shader image or buffer variable stores or atomic operations must be identical each time the command is executed on that initial Vulkan and framebuffer state.

Rule 7 Identical pipelines will produce the same result when run multiple times with the same input.
as long as:

- shader invocations do not use image atomic operations;
- no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and
- no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.

Note

The OpenGL spec has the following invariance rule: Consider a primitive p' obtained by translating a primitive p through an offset (x, y) in window coordinates, where x and y are integers. As long as neither p' nor p is clipped, it must be the case that each fragment f' produced from p' is identical to a corresponding fragment f from p except that the center of f' is offset by (x, y) from the center of f.

This rule does not apply to Vulkan and is an intentional difference from OpenGL.

When any sequence of Vulkan commands triggers shader invocations that perform image stores or atomic operations, and subsequent Vulkan commands read the memory written by those shader invocations, these operations must be explicitly synchronized.

Tessellation Invariance

When using a pipeline containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding “cracks” caused by minor differences in the positions of vertices along shared edges.

Rule 1 When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the pipeline used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode decorations. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered m of the primitive numbered n are identical for all values of m and n.

Rule 2 The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depend only on the corresponding outer tessellation level and the spacing decorations in the tessellation shaders of the pipeline.

Rule 3 The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form (0, x, 1-x), (x, 0, 1-x), or (x, 1-x, 0), it will also generate a vertex with coordinates of exactly (0, 1-x, x), (1-x, 0, x), or (1-x, x, 0), respectively. For quad tessellation, if the subdivision generates a vertex with
coordinates of $(x, 0)$ or $(0, x)$, it will also generate a vertex with coordinates of exactly $(1-x, 0)$ or $(0, 1-x)$, respectively. For isoline tessellation, if it generates vertices at $(0, x)$ and $(1, x)$ where $x$ is not zero, it will also generate vertices at exactly $(0, 1-x)$ and $(1, 1-x)$, respectively.

**Rule 4** The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at $(x, 1-x, 0)$ and $(1-x, x, 0)$ are generated when subdividing the $w = 0$ edge in triangular tessellation, vertices must be generated at $(x, 0, 1-x)$ and $(1-x, 0, x)$ when subdividing an otherwise identical $v = 0$ edge. For quad tessellation, if vertices at $(x, 0)$ and $(1-x, 0)$ are generated when subdividing the $v = 0$ edge, vertices must be generated at $(0, x)$ and $(0, 1-x)$ when subdividing an otherwise identical $u = 0$ edge.

**Rule 5** When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle $n_1$ produced by processing the first patch, there must be a triangle $n_2$ produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in $n_1$.

**Rule 6** When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation must be identical in all respects except for vertex and triangle order. For each interior triangle $n_1$ produced by processing the first patch, there must be a triangle $n_2$ produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in $n_1$. A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.

**Rule 7** For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing decorations.

**Rule 8** The value of all defined components of $\text{TessCoord}$ will be in the range $[0, 1]$. Additionally, for any defined component $x$ of $\text{TessCoord}$, the results of computing $1.0-x$ in a tessellation evaluation shader will be exact. If any floating-point values in the range $[0, 1]$ fail to satisfy this property, such values must not be used as tessellation coordinate components.
Appendix H: Lexicon

This appendix defines terms, abbreviations, and API prefixes used in the Specification.

Glossary

The terms defined in this section are used consistently throughout the Specification and may be used with or without capitalization.

Accessible (Descriptor Binding)
A descriptor binding is accessible to a shader stage if that stage is included in the `stageFlags` of the descriptor binding. Descriptors using that binding can only be used by stages in which they are accessible.

Acquire Operation (Resource)
An operation that acquires ownership of an image subresource or buffer range.

Active (Descriptor Type)
When a descriptor with `mutable` type is updated with `vkUpdateDescriptorSets`, the active descriptor type changes. When the descriptor is consumed by shaders, it is the active descriptor type which determines validity, i.e. `VkDescriptorSetLayoutBinding::descriptorType` is replaced with the active descriptor type. A mismatch in active descriptor type and consumption by shader is considered an undefined descriptor.

Active (Transform Feedback)
Transform feedback is made active after `vkCmdBeginTransformFeedbackEXT` executes and remains active until `vkCmdEndTransformFeedbackEXT` executes. While transform feedback is active, data written to variables in the output interface of the last vertex processing stage of the graphics pipeline are captured to the bound transform feedback buffers if those variables are decorated for transform feedback.

Adjacent Vertex
A vertex in an adjacency primitive topology that is not part of a given primitive, but is accessible in geometry shaders.

Active Object (Ray Tracing)
A primitive or instance in a ray tracing acceleration structure which has a corresponding ID, and is not `inactive` (meaning that it is visible to rays).

Advanced Blend Operation
Blending performed using one of the blend operation enums introduced by the `VK_EXT_blend_operation_advanced` extension. See Advanced Blending Operations.

Alias (API type/command)
An identical definition of another API type/command with the same behavior but a different name.
Aliased Range (Memory)
A range of a device memory allocation that is bound to multiple resources simultaneously.

Allocation Scope
An association of a host memory allocation to a parent object or command, where the allocation’s lifetime ends before or at the same time as the parent object is freed or destroyed, or during the parent command.

Aspect (Image)
An image may contain multiple kinds, or aspects, of data for each pixel, where each aspect is used in a particular way by the pipeline and may be stored differently or separately from other aspects. For example, the color components of an image format make up the color aspect of the image, and may be used as a framebuffer color attachment. Some operations, like depth testing, operate only on specific aspects of an image. Other operations, like image/buffer copies, only operate on one aspect at a time.

Attachment (Render Pass)
A zero-based integer index name used in render pass creation to refer to a framebuffer attachment that is accessed by one or more subpasses. The index also refers to an attachment description which includes information about the properties of the image view that will later be attached.

Availability Operation
An operation that causes the values generated by specified memory write accesses to become available for future access.

Available
A state of values written to memory that allows them to be made visible.

Axis-aligned Bounding Box
A box bounding a region in space defined by extents along each axis and thus representing a box where each edge is aligned to one of the major axes.

Back-Facing
See Facingness.

Batch
A single structure submitted to a queue as part of a queue submission command, describing a set of queue operations to execute.

Backwards Compatibility
A given version of the API is backwards compatible with an earlier version if an application, relying only on valid behavior and functionality defined by the earlier specification, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

Binary Semaphore
A semaphore with a boolean payload indicating whether the semaphore is signaled or
unsigned. Represented by a `VkSemaphore` object created with a semaphore type of `VK_SEMAPHORE_TYPE_BINARY`.

**Binding (Memory)**

An association established between a range of a resource object and a range of a memory object. These associations determine the memory locations affected by operations performed on elements of a resource object. Memory bindings are established using the `vkBindBufferMemory` command for non-sparse buffer objects, using the `vkBindImageMemory` command for non-sparse image objects, and using the `vkQueueBindSparse` command for sparse resources.

**Blend Constant**

Four floating point (RGBA) values used as an input to blending.

**Blending**

Arithmetic operations between a fragment color value and a value in a color attachment that produce a final color value to be written to the attachment.

**Buffer**

A resource that represents a linear array of data in device memory. Represented by a `VkBuffer` object.

**Buffer Device Address**

A 64-bit value used in a shader to access buffer memory through the `PhysicalStorageBuffer` storage class.

**Buffer View**

An object that represents a range of a specific buffer, and state that controls how the contents are interpreted. Represented by a `VkBufferView` object.

**Built-In Variable**

A variable decorated in a shader, where the decoration makes the variable take values provided by the execution environment or values that are generated by fixed-function pipeline stages.

**Built-In Interface Block**

A block defined in a shader that contains only variables decorated with built-in decorations, and is used to match against other shader stages.

**Clip Coordinates**

The homogeneous coordinate space that vertex positions (`Position` decoration) are written in by vertex processing stages.

**Clip Distance**

A built-in output from vertex processing stages that defines a clip half-space against which the primitive is clipped.

**Clip Volume**

The intersection of the view volume with all clip half-spaces.
**Color Attachment**
A subpass attachment point, or image view, that is the target of fragment color outputs and blending.

**Color Fragment**
A unique color value within a pixel of a multisampled color image. The *fragment mask* will contain indices to the *color fragment*.

**Color Renderable Format**
A `VkFormat` where `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` is set in one of the following, depending on the image's tiling:

- `VkFormatProperties::linearTilingFeatures`
- `VkFormatProperties::optimalTilingFeatures`
- `VkDrmFormatModifierPropertiesEXT::drmFormatModifierTilingFeatures`

**Combined Image Sampler**
A descriptor type that includes both a sampled image and a sampler.

**Command Buffer**
An object that records commands to be submitted to a queue. Represented by a `VkCommandBuffer` object.

**Command Pool**
An object that command buffer memory is allocated from, and that owns that memory. Command pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a `VkCommandPool` object.

**Compatible Allocator**
When allocators are compatible, allocations from each allocator can be freed by the other allocator.

**Compatible Image Formats**
When formats are compatible, images created with one of the formats can have image views created from it using any of the compatible formats. Also see *Size-Compatible Image Formats*.

**Compatible Queues**
Queues within a queue family. Compatible queues have identical properties.

**Complete Mipmap Chain**
The entire set of miplevels that can be provided for an image, from the largest application specified miplevel size down to the *minimum mipmap size*. See *Image Mipmap Sizing*.

**Completed Operation**
A deferred operation whose corresponding command has been executed to completion. See *Deferred Host Operations*
Component (Format)
A distinct part of a format. Depth, stencil, and color channels (e.g. R, G, B, A), are all separate components.

Compressed Texel Block
An element of an image having a block-compressed format, comprising a rectangular block of texel values that are encoded as a single value in memory. Compressed texel blocks of a particular block-compressed format have a corresponding width, height, and depth that define the dimensions of these elements in units of texels, and a size in bytes of the encoding in memory.

Constant Integral Expressions
A SPIR-V constant instruction whose type is `OpTypeInt`. See Constant Instruction in section 2.2.1 “Instructions” of the Khronos SPIR-V Specification.

Cooperative Matrix
A SPIR-V type where the storage for and computations performed on the matrix are spread across a set of invocations such as a subgroup.

Corner-Sampled Image
A `VkImage` where unnormalized texel coordinates are centered on integer values instead of half-integer values. Specified by setting the `VK_IMAGE_CREATE_CORNER_SAMPLED_BIT_NV` bit on `VkImageCreateInfo::flags` at image creation.

Coverage Index
The index of a sample in the coverage mask.

Coverage Mask
A bitfield associated with a fragment representing the samples that were determined to be covered based on the result of rasterization, and then subsequently modified by fragment operations or the fragment shader.

Cull Distance
A built-in output from vertex processing stages that defines a cull half-space where the primitive is rejected if all vertices have a negative value for the same cull distance.

Cull Volume
The intersection of the view volume with all cull half-spaces.

Decoration (SPIR-V)
Auxiliary information such as built-in variables, stream numbers, invariance, interpolation type, relaxed precision, etc., added to variables or structure-type members through decorations.

Deferrable Command
A command which allows deferred execution of host-side work. See Deferred Host Operations.

Deferrable Operation
A single logical item of host-side work which can be deferred. Represented by the
A feature is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose.

A subpass attachment point, or image view, that is the target of depth and/or stencil test operations and writes.

A VkFormat that includes depth and/or stencil components.

A VkImage (or VkImageView) with a depth/stencil format.

A subpass attachment point, or image view, that is the target of a multisample resolve operation from the corresponding depth/stencil attachment at the end of the subpass.

A set of fragment or compute shader invocations that cooperate to compute derivatives, including implicit derivatives for sampled image operations.

Information about a resource or resource view written into a descriptor set that is used to access the resource or view from a shader.

An entry in a descriptor set layout corresponding to zero or more descriptors of a single descriptor type in a set. Defined by a VkDescriptorSetLayoutBinding structure.

An object that descriptor sets are allocated from, and that owns the storage of those descriptor sets. Descriptor pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a VkDescriptorPool object.

An object that resource descriptors are written into via the API, and that can be bound to a command buffer such that the descriptors contained within it can be accessed from shaders. Represented by a VkDescriptorSet object.

An object that defines the set of resources (types and counts) and their relative arrangement (in the binding namespace) within a descriptor set. Used when allocating descriptor sets and when creating pipeline layouts. Represented by a VkDescriptorSetLayout object.
Device
The processor(s) and execution environment that perform tasks requested by the application via the Vulkan API.

Device Group
A set of physical devices that support accessing each other's memory and recording a single command buffer that can be executed on all the physical devices.

Device Index
A zero-based integer that identifies one physical device from a logical device. A device index is valid if it is less than the number of physical devices in the logical device.

Device Mask
A bitmask where each bit represents one device index. A device mask value is valid if every bit that is set in the mask is at a bit position that is less than the number of physical devices in the logical device.

Device Memory
Memory accessible to the device. Represented by a VkDeviceMemory object.

Device-Level Command
Any command that is dispatched from a logical device, or from a child object of a logical device.

Device-Level Functionality
All device-level commands and objects, and their structures, enumerated types, and enumerants.

Device-Level Object
Logical device objects and their child objects. For example, VkDevice, VkQueue, and VkCommandBuffer objects are device-level objects.

Device-Local Memory
Memory that is connected to the device, and may be more performant for device access than host-local memory.

Direct Drawing Commands
Drawing commands that take all their parameters as direct arguments to the command (and not sourced via structures in buffer memory as the indirect drawing commands). Includes vkCmdDrawMeshTasksNV, vkCmdDraw, and vkCmdDrawIndexed.

Disjoint
Disjoint planes are image planes to which memory is bound independently. A disjoint image consists of multiple disjoint planes, and is created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set.

Dispatchable Handle
A handle of a pointer handle type which may be used by layers as part of intercepting API commands. The first argument to each Vulkan command is a dispatchable handle type.
Dispatching Commands
Commands that provoke work using a compute pipeline. Includes `vkCmdDispatch` and `vkCmdDispatchIndirect`.

Drawing Commands

Duration (Command)
The duration of a Vulkan command refers to the interval between calling the command and its return to the caller.

Dynamic Storage Buffer
A storage buffer whose offset is specified each time the storage buffer is bound to a command buffer via a descriptor set.

Dynamic Uniform Buffer
A uniform buffer whose offset is specified each time the uniform buffer is bound to a command buffer via a descriptor set.

Dynamically Uniform
See Dynamically Uniform in section 2.2 “Terms” of the Khronos SPIR-V Specification.

Element
Arrays are composed of multiple elements, where each element exists at a unique index within that array. Used primarily to describe data passed to or returned from the Vulkan API.

Explicitly-Enabled Layer
A layer enabled by the application by adding it to the enabled layer list in `vkCreateInstance` or `vkCreateDevice`.

Event
A synchronization primitive that is signaled when execution of previous commands completes through a specified set of pipeline stages. Events can be waited on by the device and polled by the host. Represented by a VkEvent object.

Executable State (Command Buffer)
A command buffer that has ended recording commands and can be executed. See also Initial State and Recording State.

Execution Dependency
A dependency that guarantees that certain pipeline stages’ work for a first set of commands has completed execution before certain pipeline stages’ work for a second set of commands begins execution. This is accomplished via pipeline barriers, subpass dependencies, events, or implicit ordering operations.
Execution Dependency Chain
A sequence of execution dependencies that transitively act as a single execution dependency.

Explicit chroma reconstruction
An implementation of sampler $Y'C'B'C$ conversion which reconstructs reduced-resolution chroma samples to luma resolution and then separately performs texture sample interpolation. This is distinct from an implicit implementation, which incorporates chroma sample reconstruction into texture sample interpolation.

Extension Scope
The set of objects and commands that can be affected by an extension. Extensions are either device scope or instance scope.

Extending Structure
A structure type which may appear in the pNext chain of another structure, extending the functionality of the other structure. Extending structures may be defined by either core API versions or extensions.

External Handle
A resource handle which has meaning outside of a specific Vulkan device or its parent instance. External handles may be used to share resources between multiple Vulkan devices in different instances, or between Vulkan and other APIs. Some external handle types correspond to platform-defined handles, in which case the resource may outlive any particular Vulkan device or instance and may be transferred between processes, or otherwise manipulated via functionality defined by the platform for that handle type.

External synchronization
A type of synchronization required of the application, where parameters defined to be externally synchronized must not be used simultaneously in multiple threads.

Facingness (Polygon)
A classification of a polygon as either front-facing or back-facing, depending on the orientation (winding order) of its vertices.

Facingness (Fragment)
A fragment is either front-facing or back-facing, depending on the primitive it was generated from. If the primitive was a polygon (regardless of polygon mode), the fragment inherits the facingness of the polygon. All other fragments are front-facing.

Fence
A synchronization primitive that is signaled when a set of batches or sparse binding operations complete execution on a queue. Fences can be waited on by the host. Represented by a VkFence object.

Flat Shading
A property of a vertex attribute that causes the value from a single vertex (the provoking vertex) to be used for all vertices in a primitive, and for interpolation of that attribute to return that single value unaltered.
Format Features
A set of features from VkFormatFeatureFlagBits that a VkFormat is capable of using for various commands. The list is determined by factors such as VkImageTiling.

Fragment
A rectangular framebuffer region with associated data produced by rasterization and processed by fragment operations including the fragment shader.

Fragment Area
The width and height, in pixels, of a fragment.

Fragment Density
The ratio of fragments per framebuffer area in the x and y direction.

Fragment Density Texel Size
The (w,h) framebuffer region in pixels that each texel in a fragment density map applies to.

Fragment Input Attachment Interface
Variables with UniformConstant storage class and a decoration of InputAttachmentIndex that are statically used by a fragment shader’s entry point, which receive values from input attachments.

Fragment Mask
A lookup table that associates color samples with color fragment values.

Fragment Output Interface
A fragment shader entry point's variables with Output storage class, which output to color and/or depth/stencil attachments.

Framebuffer
A collection of image views and a set of dimensions that, in conjunction with a render pass, define the inputs and outputs used by drawing commands. Represented by a VkFramebuffer object.

Framebuffer Attachment
One of the image views used in a framebuffer.

Framebuffer Coordinates
A coordinate system in which adjacent pixels’ coordinates differ by 1 in x and/or y, with (0,0) in the upper left corner and pixel centers at half-integers.

Framebuffer-Space
Operating with respect to framebuffer coordinates.

Framebuffer-Local
A framebuffer-local dependency guarantees that only for a single framebuffer region, the first set of operations happens-before the second set of operations.
Framebuffer-Global
A framebuffer-global dependency guarantees that for all framebuffer regions, the first set of operations happens-before the second set of operations.

Framebuffer Region
A framebuffer region is a set of sample (x, y, layer, sample) coordinates that is a subset of the entire framebuffer.

Front-Facing
See Facingness.

Full Compatibility
A given version of the API is fully compatible with another version if an application, relying only on valid behavior and functionality defined by either of those specifications, is able to correctly run against each version without any modification. This assumes no active attempt by that application to not run when it detects a different version.

Global Workgroup
A collection of local workgroups dispatched by a single dispatch command. In addition to the compute dispatch, a single mesh task draw command can also generate such a collection.

Handle
An opaque integer or pointer value used to refer to a Vulkan object. Each object type has a unique handle type.

Happen-after, happens-after
A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of A and a destination of B enforces that B happens-after A. The inverse relation of happens-before.

Happen-before, happens-before
A transitive, irreflexive and antisymmetric ordering relation between operations. An execution dependency with a source of A and a destination of B enforces that A happens-before B. The inverse relation of happens-after.

Helper Invocation
A fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations, and which does not have side effects.

Host
The processor(s) and execution environment that the application runs on, and that the Vulkan API is exposed on.

Host Mapped Device Memory
Device memory that is mapped for host access using 

Host Mapped Foreign Memory
Memory owned by a foreign device that is mapped for host access.
Host Memory
Memory not accessible to the device, used to store implementation data structures.

Host-Accessible Subresource
A buffer, or a linear image subresource in either the `VK_IMAGE_LAYOUT_PREINITIALIZED` or `VK_IMAGE_LAYOUT_GENERAL` layout. Host-accessible subresources have a well-defined addressing scheme which can be used by the host.

Host-Local Memory
Memory that is not local to the device, and may be less performant for device access than device-local memory.

Host-Visible Memory
Device memory that can be mapped on the host and can be read and written by the host.

Identically Defined Objects
Objects of the same type where all arguments to their creation or allocation functions, with the exception of `pAllocator`, are

1. Vulkan handles which refer to the same object or
2. identical scalar or enumeration values or
3. Host pointers which point to an array of values or structures which also satisfy these three constraints.

Image
A resource that represents a multi-dimensional formatted interpretation of device memory. Represented by a `VkImage` object.

Image Subresource
A specific mipmap level and layer of an image.

Image Subresource Range
A set of image subresources that are contiguous mipmap levels and layers.

Image View
An object that represents an image subresource range of a specific image, and state that controls how the contents are interpreted. Represented by a `VkImageView` object.

Immutable Sampler
A sampler descriptor provided at descriptor set layout creation time, and that is used for that binding in all descriptor sets allocated from the layout, and cannot be changed.

Implicit chroma reconstruction
An implementation of sampler $YC_bC_r$ conversion which reconstructs the reduced-resolution chroma samples directly at the sample point, as part of the normal texture sampling operation. This is distinct from an explicit chroma reconstruction implementation, which reconstructs the reduced-resolution chroma samples to the resolution of the luma samples, then filters the result.
as part of texture sample interpolation.

**Implicitly-Enabled Layer**
A layer enabled by a loader-defined mechanism outside the Vulkan API, rather than explicitly by the application during instance or device creation.

**Inactive Object (Ray Tracing)**
A primitive or instance in a ray tracing acceleration structure which has a corresponding ID, but which will never report an intersection with any ray.

**Index Buffer**
A buffer bound via `vkCmdBindIndexBuffer` which is the source of index values used to fetch vertex attributes for a `vkCmdDrawIndexed` or `vkCmdDrawIndexedIndirect` command.

**Indexed Drawing Commands**
*Drawing commands* which use an *index buffer* as the source of index values used to fetch vertex attributes for a drawing command. Includes `vkCmdDrawIndexed`, `vkCmdDrawIndexedIndirectCountKHR`, `vkCmdDrawIndexedIndirectCountAMD`, and `vkCmdDrawIndexedIndirect`.

**Indirect Commands**
Drawing or dispatching commands that source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`, `vkCmdDrawIndexedIndirectCountKHR`, `vkCmdDrawIndexedIndirectCountAMD`, `vkCmdDrawMeshTasksIndirectNV`, `vkCmdDrawMeshTasksIndirectCountNV`, and `vkCmdDispatchIndirect`.

**Indirect Commands Layout**
A definition of a sequence of commands, that are generated on the device via `vkCmdPreprocessGeneratedCommandsNV` and `vkCmdExecuteGeneratedCommandsNV`. Each sequence is comprised of multiple `VkIndirectCommandsTokenTypeNV`, which represent a subset of traditional command buffer commands. Represented as `VkIndirectCommandsLayoutNV`.

**Indirect Drawing Commands**
*Drawing commands* that source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, `vkCmdDrawIndirectCountKHR`, `vkCmdDrawIndexedIndirectCountKHR`, `vkCmdDrawIndexedIndirectCountAMD`, `vkCmdDrawMeshTasksIndirectNV`, `vkCmdDrawMeshTasksIndirectCountNV`, and `vkCmdDrawIndexedIndirect`.

**Initial State (Command Buffer)**
A command buffer that has not begun recording commands. See also Recording State and Executable State.

**Inline Uniform Block**
A descriptor type that represents uniform data stored directly in descriptor sets, and supports read-only access in a shader.
Input Attachment

A descriptor type that represents an image view, and supports unfiltered read-only access in a shader, only at the fragment’s location in the view.

Instance

The top-level Vulkan object, which represents the application’s connection to the implementation. Represented by a VkInstance object.

Instance-Level Command

Any command that is dispatched from an instance, or from a child object of an instance, except for physical devices and their children.

Instance-Level Functionality

All instance-level commands and objects, and their structures, enumerated types, and enumerants.

Instance-Level Object

High-level Vulkan objects, which are not physical devices, nor children of physical devices. For example, VkInstance is an instance-level object.

Instance (Memory)

In a logical device representing more than one physical device, some device memory allocations have the requested amount of memory allocated multiple times, once for each physical device in a device mask. Each such replicated allocation is an instance of the device memory.

Instance (Resource)

In a logical device representing more than one physical device, buffer and image resources exist on all physical devices but can be bound to memory differently on each. Each such replicated resource is an instance of the resource.

Internal Synchronization

A type of synchronization required of the implementation, where parameters not defined to be externally synchronized may require internal mutexing to avoid multithreaded race conditions.

Invocation (Shader)

A single execution of an entry point in a SPIR-V module. For example, a single vertex’s execution of a vertex shader or a single fragment’s execution of a fragment shader.

Invocation Group

A set of shader invocations that are executed in parallel and that must execute the same control flow path in order for control flow to be considered dynamically uniform.

Invocation Repack Instruction

A ray tracing shader call instruction where the implementation may change the set of invocations that are executing.

Join (Deferred Host Operations)

The act of instructing a thread to participate in the execution of a deferred operation. See
Deferred Host Operations.

Linear Resource

A resource is linear if it is one of the following:

- a VkBuffer
- a VkImage created with VK_IMAGE_TILING_LINEAR
- a VkImage created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and whose Linux DRM format modifier is DRM_FORMAT_MOD_LINEAR
- a VkAccelerationStructureNV

Because a VkAccelerationStructureKHR resource does not have memory bound to it directly, it is considered neither linear nor non-linear. However, the VkBuffer on which a VkAccelerationStructureKHR resource is placed is a linear resource.

A resource is non-linear if it is one of the following:

- a VkImage created with VK_IMAGE_TILING_OPTIMAL
- a VkImage created with VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT and whose Linux DRM format modifier is not DRM_FORMAT_MOD_LINEAR

Linux DRM Format Modifier

A 64-bit, vendor-prefixed, semi-opaque unsigned integer describing vendor-specific details of an image's memory layout. In Linux graphics APIs, modifiers are commonly used to specify the memory layout of externally shared images. An image has a modifier if and only if it is created with tiling equal to VK_IMAGE_TILING_DRM_FORMAT_MODIFIER_EXT. For more details, refer to the appendix for extension VK_EXT_image_drm_format_modifier.

Local Workgroup

A collection of compute shader invocations invoked by a single dispatch command, which share data via WorkgroupLocal variables and can synchronize with each other.

Logical Device

An object that represents the application’s interface to the physical device. The logical device is the parent of most Vulkan objects. Represented by a VkDevice object.

Logical Operation

Bitwise operations between a fragment color value and a value in a color attachment, that produce a final color value to be written to the attachment.

Lost Device

A state that a logical device may be in as a result of unrecoverable implementation errors, or other exceptional conditions.

Mappable

See Host-Visible Memory.
Memory Dependency
A memory dependency is an execution dependency which includes availability and visibility operations such that:

- The first set of operations happens-before the availability operation
- The availability operation happens-before the visibility operation
- The visibility operation happens-before the second set of operations

Memory Domain
A memory domain is an abstract place to which memory writes are made available by availability operations and memory domain operations. The memory domains correspond to the set of agents that the write can then be made visible to. The memory domains are host, device, shader, workgroup instance (for workgroup instance there is a unique domain for each compute workgroup) and subgroup instance (for subgroup instance there is a unique domain for each subgroup).

Memory Domain Operation
An operation that makes the writes that are available to one memory domain available to another memory domain.

Memory Heap
A region of memory from which device memory allocations can be made.

Memory Type
An index used to select a set of memory properties (e.g. mappable, cached) for a device memory allocation.

Mesh Shading Pipeline
A graphics pipeline where the primitives are assembled explicitly in the shader stages. In contrast to the primitive shading pipeline where input primitives are assembled by fixed function processing.

Mesh Tasks Drawing Commands
Drawing commands which create shader invocations organized in workgroups for drawing mesh tasks. Includes vkCmdDrawMeshTasksNV, vkCmdDrawMeshTasksIndirectNV, and vkCmdDrawMeshTasksIndirectCountNV.

Minimum Miplevel Size
The smallest size that is permitted for a miplevel. For conventional images this is 1x1x1. For corner-sampled images, this is 2x2x2. See Image Miplevel Sizing.

Mip Tail Region
The set of mipmap levels of a sparse residency texture that are too small to fill a sparse block, and that must all be bound to memory collectively and opaquely.

Multi-planar
A multi-planar format (or “planar format”) is an image format consisting of more than one plane,
identifiable with a `_2PLANE` or `_3PLANE` component to the format name and listed in Formats requiring sampler Y’CbCr conversion for VK_IMAGE_ASPECT_COLOR_BIT image views. A multi-planar image (or “planar image”) is an image of a multi-planar format.

**Non-Dispatchable Handle**

A handle of an integer handle type. Handle values may not be unique, even for two objects of the same type.

**Non-Indexed Drawing Commands**

_Drawing commands_ for which the vertex attributes are sourced in linear order from the vertex input attributes for a drawing command (i.e. they do not use an _index buffer_). Includes vkCmdDraw, vkCmdDrawIndirectCountKHR, vkCmdDrawIndirectCountAMD, and vkCmdDrawIndirect.

**Normalized**

A value that is interpreted as being in the range [0,1] as a result of being implicitly divided by some other value.

**Normalized Device Coordinates**

A coordinate space after perspective division is applied to clip coordinates, and before the viewport transformation converts to framebuffer coordinates.

**Obsoleted (feature)**

A feature is obsolete if it can no longer be used.

**Opaque Capture Address**

A 64-bit value representing the device address of a buffer or memory object that is expected to be used by trace capture/replay tools in combination with the bufferDeviceAddress feature.

**Overlapped Range (Aliased Range)**

The aliased range of a device memory allocation that intersects a given image subresource of an image or range of a buffer.

**Ownership (Resource)**

If an entity (e.g. a queue family) has ownership of a resource, access to that resource is well-defined for access by that entity.

**Packed Format**

A format whose components are stored as a single texel block in memory, with their relative locations defined within that element.

**Passthrough Geometry Shader**

A geometry shader which uses the PassthroughNV decoration on a variable in its input interface. Output primitives in a passthrough geometry shader always have the same topology as the input primitive and are not produced by emitting vertices.

**Payload**

Importable or exportable reference to the internal data of an object in Vulkan.
Per-View
A variable that has an array of values which are output, one for each view that is being generated. A mesh shader which uses the `PerViewNV` decoration on a variable in its output interface.

Peer Memory
An instance of memory corresponding to a different physical device than the physical device performing the memory access, in a logical device that represents multiple physical devices.

Physical Device
An object that represents a single device in the system. Represented by a `VkPhysicalDevice` object.

Physical-Device-Level Command
Any command that is dispatched from a physical device.

Physical-Device-Level Functionality
All physical-device-level commands and objects, and their structures, enumerated types, and enumerants.

Physical-Device-Level Object
Physical device objects. For example, `VkPhysicalDevice` is a physical-device-level object.

Pipeline
An object that controls how graphics or compute work is executed on the device. A pipeline includes one or more shaders, as well as state controlling any non-programmable stages of the pipeline. Represented by a `VkPipeline` object.

Pipeline Barrier
An execution and/or memory dependency recorded as an explicit command in a command buffer, that forms a dependency between the previous and subsequent commands.

Pipeline Cache
An object that can be used to collect and retrieve information from pipelines as they are created, and can be populated with previously retrieved information in order to accelerate pipeline creation. Represented by a `VkPipelineCache` object.

Pipeline Layout
An object that defines the set of resources (via a collection of descriptor set layouts) and push constants used by pipelines that are created using the layout. Used when creating a pipeline and when binding descriptor sets and setting push constant values. Represented by a `VkPipelineLayout` object.

Pipeline Library
A pipeline that cannot be directly used, instead defining a set of shaders and shader groups which will be linked into other pipelines.
Pipeline Stage
A logically independent execution unit that performs some of the operations defined by an action command.

pNext Chain
A set of structures chained together through their pNext members.

Planar
See multi-planar.

Plane
An image plane is part of the representation of an image, containing a subset of the color channels required to represent the texels in the image and with a contiguous mapping of coordinates to bound memory. Most images consist only of a single plane, but some formats spread the channels across multiple image planes. The host-accessible properties of each image plane are accessed in a linear layout using vkGetImageSubresourceLayout. If a multi-planar image is created with the VK_IMAGE_CREATE_DISJOINT_BIT bit set, the image is described as disjoint, and its planes are therefore bound to memory independently.

Point Sampling (Rasterization)
A rule that determines whether a fragment sample location is covered by a polygon primitive by testing whether the sample location is in the interior of the polygon in framebuffer-space, or on the boundary of the polygon according to the tie-breaking rules.

Potential Format Features
The union of all VkFormatFeatureFlagBits that the implementation supports for a specified VkFormat, over all supported image tilings. For external formats the VkFormatFeatureFlagBits is provided by the implementation.

Presentable image
A VkImage object obtained from a VkSwapchainKHR used to present to a VkSurfaceKHR object.

Preserve Attachment
One of a list of attachments in a subpass description that is not read or written by the subpass, but that is read or written on earlier and later subpasses and whose contents must be preserved through this subpass.

Primary Command Buffer
A command buffer that can execute secondary command buffers, and can be submitted directly to a queue.

Primitive Shading Pipeline
A graphics pipeline where input primitives are assembled by fixed function processing. It is the counterpart to mesh shading.

Primitive Topology
State that controls how vertices are assembled into primitives, e.g. as lists of triangles, strips of lines, etc.
**Promoted (feature)**

A feature from an older extension is considered promoted if it is made available as part of a new core version or newer extension with wider support.

**Protected Buffer**

A buffer to which protected device memory can be bound.

**Protected-capable Device Queue**

A device queue to which protected command buffers can be submitted.

**Protected Command Buffer**

A command buffer which can be submitted to a protected-capable device queue.

**Protected Device Memory**

Device memory which can be visible to the device but must not be visible to the host.

**Protected Image**

An image to which protected device memory can be bound.

**Provisional**

A feature is released provisionally in order to get wider feedback on the functionality before it is finalized. Provisional features may change in ways that break backwards compatibility, and thus are not recommended for use in production applications.

**Provoking Vertex**

The vertex in a primitive from which flat shaded attribute values are taken. This is generally the “first” vertex in the primitive, and depends on the primitive topology.

**Push Constants**

A small bank of values writable via the API and accessible in shaders. Push constants allow the application to set values used in shaders without creating buffers or modifying and binding descriptor sets for each update.

**Push Constant Interface**

The set of variables with PushConstant storage class that are statically used by a shader entry point, and which receive values from push constant commands.

**Push Descriptors**

Descriptors that are written directly into a command buffer rather than into a descriptor set. Push descriptors allow the application to set descriptors used in shaders without allocating or modifying descriptor sets for each update.

**Descriptor Update Template**

An object that specifies a mapping from descriptor update information in host memory to elements in a descriptor set, which helps enable more efficient descriptor set updates.

**Query Pool**

An object containing a number of query entries and their associated state and results.
Represented by a `VkQueryPool` object.

Queue
An object that executes command buffers and sparse binding operations on a device. Represented by a `VkQueue` object.

Queue Family
A set of queues that have common properties and support the same functionality, as advertised in `VkQueueFamilyProperties`.

Queue Operation
A unit of work to be executed by a specific queue on a device, submitted via a queue submission command. Each queue submission command details the specific queue operations that occur as a result of calling that command. Queue operations typically include work that is specific to each command, and synchronization tasks.

Queue Submission
Zero or more batches and an optional fence to be signaled, passed to a command for execution on a queue. See the Devices and Queues chapter for more information.

Recording State (Command Buffer)
A command buffer that is ready to record commands. See also Initial State and Executable State.

Release Operation (Resource)
An operation that releases ownership of an image subresource or buffer range.

Render Pass
An object that represents a set of framebuffer attachments and phases of rendering using those attachments. Represented by a `VkRenderPass` object.

Render Pass Instance
A use of a render pass in a command buffer.

Required Extensions
Extensions that **must** be enabled alongside extensions dependent on them (see Extension Dependencies).

Reset (Command Buffer)
Resetting a command buffer discards any previously recorded commands and puts a command buffer in the initial state.

Residency Code
An integer value returned by sparse image instructions, indicating whether any sparse unbound texels were accessed.

Resolve Attachment
A subpass attachment point, or image view, that is the target of a multisample resolve operation from the corresponding color attachment at the end of the subpass.
Retired Swapchain

A swapchain that has been used as the oldSwapchain parameter to vkCreateSwapchainKHR. Images cannot be acquired from a retired swapchain, however images that were acquired (but not presented) before the swapchain was retired can be presented.

Sample Index

The index of a sample within a single set of samples.

Sample Shading

Invoking the fragment shader multiple times per fragment, with the covered samples partitioned among the invocations.

Sampled Image

A descriptor type that represents an image view, and supports filtered (sampled) and unfiltered read-only access in a shader.

Sampler

An object containing state that controls how sampled image data is sampled (or filtered) when accessed in a shader. Also a descriptor type describing the object. Represented by a VkSampler object.

Secondary Command Buffer

A command buffer that can be executed by a primary command buffer, and must not be submitted directly to a queue.

Self-Dependency

A subpass dependency from a subpass to itself, i.e. with srcSubpass equal to dstSubpass. A self-dependency is not automatically performed during a render pass instance, rather a subset of it can be performed via vkCmdPipelineBarrier during the subpass.

Semaphore

A synchronization primitive that supports signal and wait operations, and can be used to synchronize operations within a queue or across queues. Represented by a VkSemaphore object.

Shader

Instructions selected (via an entry point) from a shader module, which are executed in a shader stage.

Shader Call

An instruction which may cause execution to continue in a different shader stage.

Shader Code

A stream of instructions used to describe the operation of a shader.

Shader Group

A set of Shader Stages that are part of a VkPipeline which contains multiple of such sets. This allows the device to make use of all the shader groups from the bound pipeline independently.
Shader Module
A collection of shader code, potentially including several functions and entry points, that is used to create shaders in pipelines. Represented by a VkShaderModule object.

Shader Stage
A stage of the graphics or compute pipeline that executes shader code.

Shading Rate
The ratio of the number of fragment shader invocations generated in a fully covered framebuffer region to the size (in pixels) of that region.

Shading Rate Image
An image used to establish the shading rate for a framebuffer region, where each pixel controls the shading rate for a corresponding framebuffer region.

Shared presentable image
A presentable image created from a swapchain with VkPresentModeKHR set to either VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR or VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR.

Side Effect
A store to memory or atomic operation on memory from a shader invocation.

Single-plane format
A format that is not multi-planar.

Size-Compatible Image Formats
When a compressed image format and an uncompressed image format are size-compatible, it means that the texel block size of the uncompressed format must equal the texel block size of the compressed format.

Sparse Block
An element of a sparse resource that can be independently bound to memory. Sparse blocks of a particular sparse resource have a corresponding size in bytes that they use in the bound memory.

Sparse Image Block
A sparse block in a sparse partially-resident image. In addition to the sparse block size in bytes, sparse image blocks have a corresponding width, height, and depth that define the dimensions of these elements in units of texels or compressed texel blocks, the latter being used in case of sparse images having a block-compressed format.

Sparse Unbound Texel
A texel read from a region of a sparse texture that does not have memory bound to it.

Static Use
An object in a shader is statically used by a shader entry point if any function in the entry point’s call tree contains an instruction using the object. Static use is used to constrain the set of descriptors used by a shader entry point.
Storage Buffer
A descriptor type that represents a buffer, and supports reads, writes, and atomics in a shader.

Storage Image
A descriptor type that represents an image view, and supports unfiltered loads, stores, and atomics in a shader.

Storage Texel Buffer
A descriptor type that represents a buffer view, and supports unfiltered, formatted reads, writes, and atomics in a shader.

Subgroup
A set of shader invocations that can synchronize and share data with each other efficiently. In compute shaders, the local workgroup is a superset of the subgroup.

Subgroup Mask
A bitmask for all invocations in the current subgroup with one bit per invocation, starting with the least significant bit in the first vector component, continuing to the last bit (less than SubgroupSize) in the last required vector component.

Subpass
A phase of rendering within a render pass, that reads and writes a subset of the attachments.

Subpass Dependency
An execution and/or memory dependency between two subpasses described as part of render pass creation, and automatically performed between subpasses in a render pass instance. A subpass dependency limits the overlap of execution of the pair of subpasses, and can provide guarantees of memory coherence between accesses in the subpasses.

Subpass Description
Lists of attachment indices for input attachments, color attachments, depth/stencil attachment, resolve attachments, depth/stencil resolve, and preserve attachments used by the subpass in a render pass.

Subset (Self-Dependency)
A subset of a self-dependency is a pipeline barrier performed during the subpass of the self-dependency, and whose stage masks and access masks each contain a subset of the bits set in the identically named mask in the self-dependency.

Texel Block
A single addressable element of an image with an uncompressed VkFormat, or a single compressed block of an image with a compressed VkFormat.

Texel Block Size
The size (in bytes) used to store a texel block of a compressed or uncompressed image.

Texel Coordinate System
One of three coordinate systems (normalized, unnormalized, integer) that define how texel
coordinates are interpreted in an image or a specific mipmap level of an image.

**Timeline Semaphore**

A semaphore with a strictly increasing 64-bit unsigned integer payload indicating whether the semaphore is signaled with respect to a particular reference value. Represented by a `VkSemaphore` object created with a semaphore type of `VK_SEMAPHORE_TYPE_TIMELINE`.

**Uniform Texel Buffer**

A descriptor type that represents a buffer view, and supports unfiltered, formatted, read-only access in a shader.

**Uniform Buffer**

A descriptor type that represents a buffer, and supports read-only access in a shader.

**Units in the Last Place (ULP)**

A measure of floating-point error loosely defined as the smallest representable step in a floating-point format near a given value. For the precise definition see Precision and Operation of SPIR-V instructions or Jean-Michel Muller, “On the definition of ulp(x)”, RR-5504, INRIA. Other sources may also use the term “unit of least precision”.

**Unnormalized**

A value that is interpreted according to its conventional interpretation, and is not normalized.

**Unprotected Buffer**

A buffer to which unprotected device memory can be bound.

**Unprotected Command Buffer**

A command buffer which can be submitted to an unprotected device queue or a protected-capable device queue.

**Unprotected Device Memory**

Device memory which can be visible to the device and can be visible to the host.

**Unprotected Image**

An image to which unprotected device memory can be bound.

**User-Defined Variable Interface**

A shader entry point’s variables with Input or Output storage class that are not built-in variables.

**Vertex Input Attribute**

A graphics pipeline resource that produces input values for the vertex shader by reading data from a vertex input binding and converting it to the attribute's format.

**Vertex Stream**

A vertex stream is where the last vertex processing stage outputs vertex data, which then goes to the rasterizer, is captured to a transform feedback buffer, or both. Geometry shaders can emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams.
Validation Cache

An object that can be used to collect and retrieve validation results from the validation layers, and can be populated with previously retrieved results in order to accelerate the validation process. Represented by a VkValidationCacheEXT object.

Vertex Input Binding

A graphics pipeline resource that is bound to a buffer and includes state that affects addressing calculations within that buffer.

Vertex Input Interface

A vertex shader entry point's variables with Input storage class, which receive values from vertex input attributes.

Vertex Processing Stages

A set of shader stages that comprises the vertex shader, tessellation control shader, tessellation evaluation shader, and geometry shader stages. The task and mesh shader stages also belong to this group.

View Mask

When multiview is enabled, a view mask is a property of a subpass controlling which views the rendering commands are broadcast to.

View Volume

A subspace in homogeneous coordinates, corresponding to post-projection x and y values between -1 and +1, and z values between 0 and +1.

Viewport Transformation

A transformation from normalized device coordinates to framebuffer coordinates, based on a viewport rectangle and depth range.

Visibility Operation

An operation that causes available values to become visible to specified memory accesses.

Visible

A state of values written to memory that allows them to be accessed by a set of operations.

Common Abbreviations

The abbreviations and acronyms defined in this section are sometimes used in the Specification and the API where they are considered clear and commonplace.

Src

Source

Dst

Destination
Min
  Minimum

Max
  Maximum

Rect
  Rectangle

Info
  Information

LOD
  Level of Detail

ID
  Identifier

UUID
  Universally Unique Identifier

Op
  Operation

R
  Red color component

G
  Green color component

B
  Blue color component

A
  Alpha color component

RTZ
  Round towards zero

RTE
  Round to nearest even

Prefixes

Prefixes are used in the API to denote specific semantic meaning of Vulkan names, or as a label to avoid name clashes, and are explained here:
**VK/Vk/vk**

Vulkan namespace

All types, commands, enumerants and defines in this specification are prefixed with these two characters.

**PFN/pfn**

Function Pointer

Denotes that a type is a function pointer, or that a variable is of a pointer type.

**p**

Pointer

Variable is a pointer.

**vkCmd**

Commands that record commands in command buffers

These API commands do not result in immediate processing on the device. Instead, they record the requested action in a command buffer for execution when the command buffer is submitted to a queue.

**s**

Structure

Used to denote the `VK_STRUCTURE_TYPE*` member of each structure in `sType`
Appendix I: Credits (Informative)

Vulkan 1.2 is the result of contributions from many people and companies participating in the Khronos Vulkan Working Group, as well as input from the Vulkan Advisory Panel.

Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed in the following section. Some specific contributions made by individuals are listed together with their name.

Working Group Contributors to Vulkan

- Aaron Greig, Codeplay Software Ltd. (version 1.1)
- Aaron Hagan, AMD (version 1.1)
- Adam Jackson, Red Hat (versions 1.0, 1.1)
- Adam Śmigielski, Mobica (version 1.0)
- Aidan Fabius, Core Avionics & Industrial Inc. (version 1.2)
- Alan Baker, Google (versions 1.1, 1.2)
- Alan Ward, Google (versions 1.1, 1.2)
- Alejandro Piñeiro, Igalia (version 1.1)
- Alex Bourd, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
- Alex Crabb, Caster Communications (version 1.2)
- Alex Walters, Imagination Technologies (version 1.2)
- Alexander Galazin, Arm (versions 1.0, 1.1, 1.2)
- Allen Hux, Intel (version 1.0)
- Alon Or-bach, Samsung Electronics (versions 1.0, 1.1, 1.2) (WSI technical sub-group chair)
- Anastasia Stulova, Arm (version 1.2)
- Andreas Vasilakis, Think Silicon (version 1.2)
- Andres Gomez, Igalia (version 1.1)
- Andrew Cox, Samsung Electronics (version 1.0)
- Andrew Garrard, Samsung Electronics (versions 1.0, 1.1, 1.2) (format wrangler)
- Andrew Poole, Samsung Electronics (version 1.0)
- Andrew Rafter, Samsung Electronics (version 1.0)
- Andrew Richards, Codeplay Software Ltd. (version 1.0)
- Andrew Woloszyn, Google (versions 1.0, 1.1)
- Ann Thorsnes, Khronos (version 1.2)
- Antoine Labour, Google (versions 1.0, 1.1)
- Aras Pranckevičius, Unity Technologies (version 1.0)
• Ashwin Kolhe, NVIDIA (version 1.0)
• Baldur Karlsson, Independent (versions 1.1, 1.2)
• Barthold Lichtenbelt, NVIDIA (version 1.1)
• Bas Nieuwenhuizen, Google (versions 1.1, 1.2)
• Ben Bowman, Imagination Technologies (version 1.0)
• Benj Lipchak, Unknown (version 1.0)
• Bill Hollings, Brenwill (versions 1.0, 1.1, 1.2)
• Bill Licea-Kane, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Blaine Kohl, Khronos (version 1.2)
• Boris Zanin, Mobica (version 1.2)
• Brent E. Insko, Intel (version 1.0)
• Brian Ellis, Qualcomm Technologies, Inc. (version 1.0)
• Brian Paul, VMware (version 1.2)
• Caio Marcelo de Oliveira Filho, Intel (version 1.2)
• Cass Everitt, Oculus VR (versions 1.0, 1.1)
• Cemil Azizoglu, Canonical (version 1.0)
• Chad Versace, Google (versions 1.0, 1.1, 1.2)
• Chang-Hyo Yu, Samsung Electronics (version 1.0)
• Chia-I Wu, LunarG (version 1.0)
• Chris Frascati, Qualcomm Technologies, Inc. (version 1.0)
• Christophe Riccio, Unity Technologies (versions 1.0, 1.1)
• Cody Northrop, LunarG (version 1.0)
• Colin Riley, AMD (version 1.1)
• Cort Stratton, Google (versions 1.1, 1.2)
• Courtney Goeltzenleuchter, Google (versions 1.0, 1.1)
• Craig Davies, Huawei (version 1.2)
• Dae Kim, Imagination Technologies (version 1.1)
• Damien Leone, NVIDIA (version 1.0)
• Dan Baker, Oxide Games (versions 1.0, 1.1)
• Dan Ginsburg, Valve Software (versions 1.0, 1.1, 1.2)
• Daniel Johnston, Intel (versions 1.0, 1.1)
• Daniel Koch, NVIDIA (versions 1.0, 1.1, 1.2)
• Daniel Rakos, AMD (versions 1.0, 1.1, 1.2)
• Daniel Stone, Collabora (versions 1.1, 1.2)
• Daniel Vetter, Intel (version 1.2)
• David Airlie, Red Hat (versions 1.0, 1.1, 1.2)
• David Mao, AMD (versions 1.0, 1.2)
• David Miller, Miller & Mattson (versions 1.0, 1.1) (Vulkan reference card)
• David Neto, Google (versions 1.0, 1.1, 1.2)
• David Wilkinson, AMD (version 1.2)
• David Yu, Pixar (version 1.0)
• Dejan Mircevski, Google (version 1.1)
• Dominik Witczak, AMD (versions 1.0, 1.1)
• Donald Scorgie, Imagination Technologies (version 1.2)
• Dzmitry Malyshau, Mozilla (versions 1.1, 1.2)
• Ed Hutchins, Oculus (version 1.2)
• Emily Stearns, Khronos (version 1.2)
• Frank (LingJun) Chen, Qualcomm Technologies, Inc. (version 1.0)
• Fred Liao, Mediatek (version 1.0)
• Gabe Dagani, Freescale (version 1.0)
• Gabor Sines, AMD (version 1.2)
• Graeme Leese, Broadcom (versions 1.0, 1.1, 1.2)
• Graham Connor, Imagination Technologies (version 1.0)
• Graham Sellers, AMD (versions 1.0, 1.1)
• Greg Fischer, LunarG (version 1.1)
• Hai Nguyen, Google (version 1.2)
• Hans-Kristian Arntzen, Arm (versions 1.1, 1.2)
• Henri Verbeet, Codeweavers (version 1.2)
• Hwanyong Lee, Kyungpook National University (version 1.0)
• Iago Toral, Igalia (versions 1.1, 1.2)
• Ian Elliott, Google (versions 1.0, 1.1, 1.2)
• Ian Romanick, Intel (versions 1.0, 1.1)
• James Hughes, Oculus VR (version 1.0)
• James Jones, NVIDIA (versions 1.0, 1.1, 1.2)
• James Riordon, Khronos (version 1.2)
• Jan Hermes, Continental Corporation (versions 1.0, 1.1)
• Jan-Harald Fredriksen, Arm (versions 1.0, 1.1, 1.2)
• Jason Ekstrand, Intel (versions 1.0, 1.1, 1.2)
• Jean-François Roy, Google (versions 1.1, 1.2)
• Jeff Bolz, NVIDIA (versions 1.0, 1.1, 1.2)
Jeff Juliano, NVIDIA (versions 1.0, 1.1, 1.2)
Jeff Leger, Qualcomm Technologies, Inc. (version 1.1)
Jeff Vigil, Samsung Electronics (versions 1.0, 1.1, 1.2)
Jens Owen, Google (versions 1.0, 1.1)
Jeremy Hayes, LunarG (version 1.0)
Jesse Barker, Unity Technologies (versions 1.0, 1.1, 1.2)
Jesse Hall, Google (versions 1.0, 1.1, 1.2)
Joe Davis, Samsung Electronics (version 1.1)
Johannes van Waveren, Oculus VR (versions 1.0, 1.1)
John Kessenich, Google (versions 1.0, 1.1, 1.2) (SPIR-V and GLSL for Vulkan spec author)
John McDonald, Valve Software (versions 1.0, 1.1, 1.2)
John Zulauf, LunarG (versions 1.1, 1.2)
Jon Ashburn, LunarG (version 1.0)
Jon Leech, Independent (versions 1.0, 1.1, 1.2) (XML toolchain, normative language, release wrangler)
Jonas Gustavsson, Samsung Electronics (versions 1.0, 1.1)
Jonas Meyer, Epic Games (version 1.2)
Jonathan Hamilton, Imagination Technologies (version 1.0)
Jordan Justen, Intel (version 1.1)
Jungwoo Kim, Samsung Electronics (versions 1.0, 1.1)
Jörg Wagner, Arm (version 1.1)
Kalle Raita, Google (version 1.1)
Karen Ghavam, LunarG (versions 1.1, 1.2)
Karl Schultz, LunarG (versions 1.1, 1.2)
Kathleen Mattson, Khronos (versions 1.0, 1.1, 1.2)
Kaye Mason, Google (version 1.2)
Keith Packard, Valve (version 1.2)
Kenneth Benzie, Codeplay Software Ltd. (versions 1.0, 1.1)
Kenneth Russell, Google (version 1.1)
Kerch Holt, NVIDIA (versions 1.0, 1.1)
Kevin O’Neil, AMD (version 1.1)
Kris Rose, Khronos (version 1.2)
Kristian Kristensen, Intel (versions 1.0, 1.1)
Krzysztof Iwanicki, Samsung Electronics (version 1.0)
Larry Seiler, Intel (version 1.0)
• Lauri Ilola, Nokia (version 1.1)
• Lei Zhang, Google (version 1.2)
• Lenny Komow, LunarG (versions 1.1, 1.2)
• Lionel Landwerlin, Intel (versions 1.1, 1.2)
• Liz Maitral, Khronos (version 1.2)
• Lutz Latta, Lucasfilm (version 1.0)
• Maciej Jesionowski, AMD (version 1.1)
• Mais Alnasser, AMD (version 1.1)
• Marcin Rogucki, Mobica (version 1.1)
• Maria Rovatsou, Codeplay Software Ltd. (version 1.0)
• Mark Callow, Independent (versions 1.0, 1.1, 1.2)
• Mark Kilgard, NVIDIA (versions 1.1, 1.2)
• Mark Lobodzinski, LunarG (versions 1.0, 1.1, 1.2)
• Mark Young, LunarG (version 1.1)
• Markus Tavenrath, NVIDIA (version 1.1)
• Mateusz Przybylski, Intel (version 1.0)
• Mathias Heyer, NVIDIA (versions 1.0, 1.1)
• Mathias Schott, NVIDIA (versions 1.0, 1.1)
• Matt Netsch, Qualcomm Technologies, Inc. (version 1.1)
• Matthäus Chajdas, AMD (versions 1.1, 1.2)
• Maurice Ribble, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Maxim Lukyanov, Samsung Electronics (version 1.0)
• Michael Lentine, Google (version 1.0)
• Michael O'Hara, AMD (version 1.1)
• Michael Phillip, Samsung Electronics (version 1.2)
• Michael Wong, Codeplay Software Ltd. (version 1.1)
• Michael Worcester, Imagination Technologies (versions 1.0, 1.1)
• Michal Pietrasuik, Intel (version 1.0)
• Mika Isojarvi, Google (versions 1.0, 1.1)
• Mike Schuchardt, LunarG (versions 1.1, 1.2)
• Mike Stroyan, LunarG (version 1.0)
• Mike Weiblen, LunarG (versions 1.1, 1.2)
• Minyoung Son, Samsung Electronics (version 1.0)
• Mitch Singer, AMD (versions 1.0, 1.1, 1.2)
• Mythri Venugopal, Samsung Electronics (version 1.0)
• Naveen Leekha, Google (version 1.0)
• Neil Henning, AMD (versions 1.0, 1.1, 1.2)
• Neil Hickey, Arm (version 1.2)
• Neil Trevett, NVIDIA (versions 1.0, 1.1, 1.2)
• Nick Penwarden, Epic Games (version 1.0)
• Nicolai Hähnle, AMD (version 1.1)
• Niklas Smedberg, Unity Technologies (version 1.0)
• Norbert Nopper, Independent (versions 1.0, 1.1)
• Nuno Subtil, NVIDIA (versions 1.1, 1.2)
• Pat Brown, NVIDIA (version 1.0)
• Patrick Cozzi, Independent (version 1.1)
• Patrick Doane, Blizzard Entertainment (version 1.0)
• Peter Lohrmann, AMD (versions 1.0, 1.2)
• Petros Bantolas, Imagination Technologies (version 1.1)
• Pierre Boudier, NVIDIA (versions 1.0, 1.1, 1.2)
• Pierre-Loup Griffais, Valve Software (versions 1.0, 1.1, 1.2)
• Piers Daniell, NVIDIA (versions 1.0, 1.1, 1.2)
• Piotr Bialecki, Intel (version 1.0)
• Prabindh Sundareson, Samsung Electronics (version 1.0)
• Pyry Haulos, Google (versions 1.0, 1.1) (Vulkan conformance test subcommittee chair)
• Rajeev Rao, Qualcomm (version 1.2)
• Ralph Potter, Codeplay Software Ltd. (versions 1.1, 1.2)
• Ray Smith, Arm (versions 1.0, 1.1, 1.2)
• Richard Huddy, Samsung Electronics (version 1.2)
• Rob Barris, NVIDIA (version 1.1)
• Rob Stepinski, Transgaming (version 1.0)
• Robert Simpson, Qualcomm Technologies, Inc. (versions 1.0, 1.1)
• Rolando Caloca Olivares, Epic Games (versions 1.0, 1.1, 1.2)
• Roy Ju, Mediatek (version 1.0)
• Rufus Hamade, Imagination Technologies (version 1.0)
• Ruihao Zhang, Qualcomm Technologies, Inc. (versions 1.1, 1.2)
• Sean Ellis, Arm (version 1.0)
• Sean Harmer, KDAB Group (versions 1.0, 1.1)
• Shannon Woods, Google (versions 1.0, 1.1, 1.2)
• Slawomir Cygan, Intel (versions 1.0, 1.1)
• Slawomir Grajewski, Intel (versions 1.0, 1.1)
• Sorel Bosan, AMD (version 1.1)
• Spencer Fricke, Samsung Electronics (version 1.2)
• Stefanus Du Toit, Google (version 1.0)
• Stephen Huang, Mediatek (version 1.1)
• Steve Hill, Broadcom (versions 1.0, 1.2)
• Steve Viggers, Core Avionics & Industrial Inc. (versions 1.0, 1.2)
• Stuart Smith, Imagination Technologies (versions 1.0, 1.1, 1.2)
• Tilmann Scheller, Samsung Electronics (version 1.1)
• Tim Foley, Intel (version 1.0)
• Timo Suoranta, AMD (version 1.0)
• Timothy Lottes, AMD (versions 1.0, 1.1)
• Tobias Hector, AMD (versions 1.0, 1.1, 1.2) (validity language and toolchain)
• Tobin Ehlis, LunarG (version 1.0)
• Tom Olson, Arm (versions 1.0, 1.1, 1.2) (Working Group chair)
• Tomasz Bednarz, Independent (version 1.1)
• Tomasz Kubale, Intel (version 1.0)
• Tony Barbour, LunarG (versions 1.0, 1.1, 1.2)
• Victor Eruhimov, Unknown (version 1.1)
• Vincent Hindriksen, StreamHPC (version 1.2)
• Wayne Lister, Imagination Technologies (version 1.0)
• Wolfgang Engel, Unknown (version 1.1)
• Yanjun Zhang, VeriSilicon (versions 1.0, 1.1, 1.2)

Other Credits

The Vulkan Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

The wider Vulkan community have provided useful feedback, questions and spec changes that have helped improve the quality of the Specification via GitHub.

Administrative support to the Working Group for Vulkan 1.1 and 1.2 was provided by Khronos staff including Angela Cheng, Ann Thorsnes, Blaine Kohl, Dominic Agoro-Ombaka, Emily Stearns, Jeff Phillips, Lisie Aartsen, and Liz Maitral; and by Alex Crabb of Caster Communications.

Administrative support for Vulkan 1.0 was provided by Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, Kathleen Mattson and Michelle Clark of Gold Standard Group.

Technical support was provided by James Riordon, webmaster of Khronos.org and OpenGL.org.