Vulkan® 1.0.140 - A Specification

The Khronos® Vulkan Working Group

Table of Contents

1. Introduction ......................................................... 3
   1.1. Document Conventions .................................. 3
2. Fundamentals ...................................................... 5
   2.1. Host and Device Environment ......................... 5
   2.2. Execution Model ........................................ 5
   2.3. Object Model ........................................... 7
   2.4. Application Binary Interface ......................... 11
   2.5. Command Syntax and Duration ....................... 12
   2.6. Threading Behavior .................................. 13
   2.7. Errors .................................................. 19
   2.8. Numeric Representation and Computation ........... 27
   2.9. Fixed-Point Data Conversions ....................... 29
   2.10. Common Object Types ................................. 31
3. Initialization ...................................................... 33
   3.1. Command Function Pointers ......................... 33
   3.2. Instances ............................................. 35
4. Devices and Queues ................................................ 41
   4.1. Physical Devices ..................................... 41
   4.2. Devices ............................................... 48
   4.3. Queues ................................................. 54
5. Command Buffers .................................................. 59
   5.1. Command Buffer Lifecycle ........................... 59
   5.2. Command Pools ........................................ 61
   5.3. Command Buffer Allocation and Management ....... 66
   5.4. Command Buffer Recording ........................... 70
   5.5. Command Buffer Submission ......................... 75
   5.6. Queue Forward Progress .............................. 80
   5.7. Secondary Command Buffer Execution ................ 81
6. Synchronization and Cache Control .............................. 84
   6.1. Execution and Memory Dependencies ................ 84
   6.2. Implicit Synchronization Guarantees ................ 97
   6.3. Fences ................................................. 99
   6.4. Semaphores .......................................... 106
   6.5. Events ............................................... 109
   6.6. Pipeline Barriers .................................... 123
   6.7. Memory Barriers ..................................... 128
   6.8. Wait Idle Operations ................................ 137
   6.9. Host Write Ordering Guarantees .................... 138
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.</td>
<td>Render Pass</td>
<td>139</td>
</tr>
<tr>
<td>7.1.</td>
<td>Render Pass Creation</td>
<td>140</td>
</tr>
<tr>
<td>7.2.</td>
<td>Render Pass Compatibility</td>
<td>158</td>
</tr>
<tr>
<td>7.3.</td>
<td>Framebuffers</td>
<td>158</td>
</tr>
<tr>
<td>7.4.</td>
<td>Render Pass Commands</td>
<td>163</td>
</tr>
<tr>
<td>8.</td>
<td>Shaders</td>
<td>172</td>
</tr>
<tr>
<td>8.1.</td>
<td>Shader Modules</td>
<td>172</td>
</tr>
<tr>
<td>8.2.</td>
<td>Shader Execution</td>
<td>175</td>
</tr>
<tr>
<td>8.3.</td>
<td>Shader Memory Access Ordering</td>
<td>175</td>
</tr>
<tr>
<td>8.4.</td>
<td>Shader Inputs and Outputs</td>
<td>177</td>
</tr>
<tr>
<td>8.5.</td>
<td>Vertex Shaders</td>
<td>177</td>
</tr>
<tr>
<td>8.6.</td>
<td>Tessellation Control Shaders</td>
<td>178</td>
</tr>
<tr>
<td>8.7.</td>
<td>Tessellation Evaluation Shaders</td>
<td>178</td>
</tr>
<tr>
<td>8.8.</td>
<td>Geometry Shaders</td>
<td>179</td>
</tr>
<tr>
<td>8.9.</td>
<td>Fragment Shaders</td>
<td>179</td>
</tr>
<tr>
<td>8.10.</td>
<td>Compute Shaders</td>
<td>180</td>
</tr>
<tr>
<td>8.11.</td>
<td>Interpolation Decorations</td>
<td>180</td>
</tr>
<tr>
<td>8.12.</td>
<td>Static Use</td>
<td>181</td>
</tr>
<tr>
<td>8.13.</td>
<td>Scope</td>
<td>181</td>
</tr>
<tr>
<td>8.15.</td>
<td>Helper Invocations</td>
<td>185</td>
</tr>
<tr>
<td>9.</td>
<td>Pipelines</td>
<td>186</td>
</tr>
<tr>
<td>9.1.</td>
<td>Compute Pipelines</td>
<td>187</td>
</tr>
<tr>
<td>9.2.</td>
<td>Graphics Pipelines</td>
<td>194</td>
</tr>
<tr>
<td>9.3.</td>
<td>Pipeline destruction</td>
<td>205</td>
</tr>
<tr>
<td>9.4.</td>
<td>Multiple Pipeline Creation</td>
<td>206</td>
</tr>
<tr>
<td>9.5.</td>
<td>Pipeline Derivatives</td>
<td>206</td>
</tr>
<tr>
<td>9.6.</td>
<td>Pipeline Cache</td>
<td>207</td>
</tr>
<tr>
<td>9.7.</td>
<td>Specialization Constants</td>
<td>214</td>
</tr>
<tr>
<td>9.8.</td>
<td>Pipeline Binding</td>
<td>218</td>
</tr>
<tr>
<td>9.9.</td>
<td>Dynamic State</td>
<td>219</td>
</tr>
<tr>
<td>10.</td>
<td>Memory Allocation</td>
<td>221</td>
</tr>
<tr>
<td>10.1.</td>
<td>Host Memory</td>
<td>221</td>
</tr>
<tr>
<td>10.2.</td>
<td>Device Memory</td>
<td>228</td>
</tr>
<tr>
<td>11.</td>
<td>Resource Creation</td>
<td>245</td>
</tr>
<tr>
<td>11.1.</td>
<td>Buffers</td>
<td>245</td>
</tr>
<tr>
<td>11.2.</td>
<td>Buffer Views</td>
<td>250</td>
</tr>
<tr>
<td>11.3.</td>
<td>Images</td>
<td>253</td>
</tr>
<tr>
<td>11.4.</td>
<td>Image Layouts</td>
<td>266</td>
</tr>
<tr>
<td>11.5.</td>
<td>Image Views</td>
<td>269</td>
</tr>
<tr>
<td>11.6.</td>
<td>Resource Memory Association</td>
<td>280</td>
</tr>
</tbody>
</table>
25.2. Sample Mask Test .................................................. 530  
25.3. Multisample Coverage ............................................. 530  
25.4. Depth and Stencil Operations .................................... 531  
25.5. Depth Bounds Test .................................................. 532  
25.6. Stencil Test ............................................................ 533  
25.7. Depth Test ............................................................. 539  
25.8. Sample Counting ...................................................... 540  
25.9. Coverage Reduction .................................................. 540  
26. The Framebuffer ....................................................... 541  
26.1. Blending ................................................................. 541  
26.2. Logical Operations ................................................... 549  
26.3. Color Write Mask ..................................................... 550  
27. Dispatching Commands .................................................. 552  
28. Sparse Resources .......................................................... 559  
28.1. Sparse Resource Features ......................................... 559  
28.2. Sparse Buffers and Fully-Resident Images ...................... 560  
28.3. Sparse Partially-Resident Buffers ................................ 561  
28.4. Sparse Partially-Resident Images ................................ 561  
28.5. Sparse Memory Aliasing ............................................ 569  
28.7. Sparse Resource API ............................................... 572  
28.8. Examples ............................................................... 590  
29. Extending Vulkan ........................................................... 596  
29.1. Instance and Device Functionality ................................. 596  
29.2. Core Versions .......................................................... 596  
29.3. Layers ................................................................. 598  
29.4. Extensions ............................................................. 601  
29.5. Extension Dependencies ............................................ 605  
29.6. Compatibility Guarantees (Informative) ......................... 605  
30. Features ................................................................. 610  
30.1. Feature Requirements .............................................. 622  
31. Limits ................................................................. 623  
31.1. Limit Requirements .................................................. 634  
32. Formats ................................................................. 644  
32.1. Format Definition .................................................... 644  
32.2. Format Properties .................................................... 671  
32.3. Required Format Support .......................................... 674  
33. Additional Capabilities .................................................. 690  
33.1. Additional Image Capabilities .................................... 690  
34. Debugging ............................................................... 695  
Appendix A: Vulkan Environment for SPIR-V ........................... 697
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.

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 1. 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.

1.1. 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 Vulkan 1.1 are now defined in terms of the equivalent Vulkan 1.1 interfaces. This affects the Vulkan Specification, the Vulkan header files, and the corresponding XML Registry.

1.1.1. 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.
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.

### 1.1.2. 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).

### 1.1.3. 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 2. 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.

2.1. 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.

2.2. 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 four types of functionality that queues may support: 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.

### 2.2.1. 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.

2.3. 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, and may encode object information directly in the handle rather than acting as a reference to an underlying object. Objects of a non-dispatchable type may not have unique handle
values within a type or across types. 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.

2.3.1. 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

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
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
- VkDescriptorPool
- VkFramebuffer
- VkRenderPass
- VkCommandBuffer
- VkCommandPool
- VkDeviceMemory
- VkDescriptorSet

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 use the object:

- VkFence
- VkSemaphore
- VkCommandBuffer
- VkCommandPool

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
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
  - VkPipeline
  - VkPipelineLayout
  - VkSampler
  - VkDescriptorSetLayout
  - VkDescriptorPool
  - VkFramebuffer
  - VkRenderPass
  - VkCommandPool
  - VkCommandBuffer
• VkDeviceMemory
• 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.

2.4. 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.
2.5. 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 \texttt{<stdint.h>}), with exceptions described below, or elsewhere in the text when appropriate:

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

```c
typedef uint32_t VkBool32;
```

\texttt{VK_TRUE} represents a boolean \texttt{True} (integer 1) value, and \texttt{VK_FALSE} a boolean \texttt{False} (integer 0) value.

All values returned from a Vulkan implementation in a \texttt{VkBool32} will be either \texttt{VK_TRUE} or \texttt{VK_FALSE}.

Applications must not pass any other values than \texttt{VK_TRUE} or \texttt{VK_FALSE} into a Vulkan implementation where a \texttt{VkBool32} is expected.

\texttt{VkDeviceSize} represents device memory size and offset values:

```c
typedef uint64_t VkDeviceSize;
```

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

The last in-parameter to each command that creates or destroys a Vulkan object is \texttt{pAllocator}. The \texttt{pAllocator} parameter can be set to a non-\texttt{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 \texttt{vkAllocate*}, and take \texttt{Vk*AllocateInfo} structures. These Vulkan objects are freed with commands of the form \texttt{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 \texttt{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.
2.5.1. 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.

2.6. 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 mutexing 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.

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`

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

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 `vkCmdSetStencilReference`
• 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 `vkCmdBindVertexBuffer`
• 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`
2.7. 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.
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.

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.

### 2.7.1. Valid Usage

Valid usage defines a set of conditions which must be met in order to achieve well-defined run-time 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 run-time 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.

This does lead to some non-obvious places for valid usage statements. For instance, the valid values for a structure might depend on a separate value in the calling command. In this case, the structure itself will not reference this valid usage as it is impossible to determine validity from the structure that it is invalid - instead this valid usage would be attached to the calling command.

Another example is draw state - the state setters are independent, and can cause a legitimately invalid state configuration between draw calls; so the valid usage statements are attached to the place where all state needs to be valid - at the draw command.

Valid usage conditions are described in a block labelled “Valid Usage” following each command or structure they apply to.
2.7.2. 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`:

```c
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
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_INPUT_ASSEMBLY_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_RASTERIZATION_STATE_CREATE_INFO = 22,
    VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO = 23,
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO = 24,
    VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO = 25,
    VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO = 26,
    VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO = 27,
    VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO = 28,
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO = 29,
    VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO = 30,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO = 31,
    VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO = 32,
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO = 33,
    VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET = 34,
    VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET = 35,
    VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO = 36,
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO = 37,
    VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO = 38,
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO = 39,
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO = 40,
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO = 41,
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_CREATE_INFO = 42,
```

Chapter 2. Fundamentals | 23
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 structure defined by an extension, 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. If that extension is supported by the implementation, then it must be enabled.

Each type of valid 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 structures in the chain with `sType` values not defined by extensions supported by that component.

Extension structures are not described in the base Vulkan Specification, but either in layered Specifications incorporating those extensions, or in separate vendor-provided documents.

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
typedef struct VkBaseInStructure {
    VkStructureType sType;
    const struct VkBaseInStructure* pNext;
} VkBaseInStructure;
```

- `sType` is the structure type of the structure being iterated through.

24 | Chapter 2. Fundamentals
• 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
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`.

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**

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`.
2.7.3. 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 run time. All run time error codes are negative values.

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

```c
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,
} 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

**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_FRAGMENTS_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.

• **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 run time 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.

Out of memory errors do not damage any currently existing Vulkan objects. Objects that have already been successfully created can still be used by the application.

**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 run time error occurs in such commands, the implementation will defer reporting the error until a specified point. For commands that record into command buffers (vkCmd*) run time errors are reported by vkEndCommandBuffer.

### 2.8. 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.

### 2.8.1. 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.

### 2.8.2. 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.

2.8.3. 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.

2.8.4. 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.

2.8.5. 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.

2.8.6. 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.

2.9. 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 1-bit binary two's-
complement integers and binary unsigned integers, respectively.
2.9.1. 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

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

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

\[
f = \max\left(\frac{c}{2^b - 1} - 1.0, -1.0\right)
\]

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. Note that while zero is exactly expressible in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point.

2.9.2. 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 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 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 -(\(2^{b-1} - 1\)), 0, or \(2^{b-1} - 1\), respectively.

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point.
2.10. 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.

2.10.1. 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 offsets is defined by the structure:

```c
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
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.

2.10.2. 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
typedef struct VkExtent2D {
    uint32_t width;
    uint32_t height;
} VkExtent2D;
```
- \textit{width} is the width of the extent.
- \textit{height} is the height of the extent.

A three-dimensional extent is defined by the structure:

```c
typedef struct VkExtent3D {
    uint32_t width;
    uint32_t height;
    uint32_t depth;
} VkExtent3D;
```

- \textit{width} is the width of the extent.
- \textit{height} is the height of the extent.
- \textit{depth} is the depth of the extent.

### 2.10.3. 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
typedef struct VkRect2D {
    VkOffset2D offset;
    VkExtent2D extent;
} VkRect2D;
```

- \textit{offset} is a \texttt{VkOffset2D} specifying the rectangle offset.
- \textit{extent} is a \texttt{VkExtent2D} specifying the rectangle extent.
Chapter 3. Initialization

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

3.1. 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
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 valid</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid non-NULL instance</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>NULL</td>
<td>vkEnumerateInstanceExtensionProperties</td>
<td>fp</td>
</tr>
<tr>
<td>instance</td>
<td>pName</td>
<td>return value</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------</td>
<td>--------------</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^3</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)

- If instance is not NULL, instance must be a valid VkInstance handle
- 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:

```c
PFN_vkVoidFunction vkGetDeviceProcAddr(
    VkDevice                                    device,
    const char*                                 pName);
```

The table below defines the various use cases for vkGetDeviceProcAddr and expected return value for
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>NULL</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>invalid device</td>
<td>*1</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>NULL</td>
<td>undefined</td>
</tr>
<tr>
<td>device</td>
<td>core device-level Vulkan command</td>
<td>fp²</td>
</tr>
<tr>
<td>device</td>
<td>enabled extension device-level commands</td>
<td>fp²</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 `device` or a child of `device` e.g. `VkDevice`, `VkQueue`, or `VkCommandBuffer`.

**Valid Usage (Implicit)**

- `device` **must** be a valid `VkDevice` handle
- `pName` **must** be a null-terminated UTF-8 string

The definition of `PFN_vkVoidFunction` is:

```c
typedef void (VKAPI_PTR *PFN_vkVoidFunction)(void);
```

## 3.2. 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:
VK_DEFINE_HANDLE(VkInstance)

To create an instance object, call:

```c
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

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

### Valid Usage (Implicit)

- `pCreateInfo` must be a valid pointer to a valid `VkInstanceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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:

```c
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 an extension-specific 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)

- **sType** must be `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO`
- **pNext** must be `NULL`
- **flags** must be `0`
  - If `pApplicationInfo` is not `NULL`, `pApplicationInfo` must be a valid pointer to a valid `VkApplicationInfo` structure
  - If `enabledLayerCount` is not `0`, `ppEnabledLayerNames` must be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings
  - If `enabledExtensionCount` is not `0`, `ppEnabledExtensionNames` must be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings

```c
typedef VkFlags VkInstanceCreateFlags;
```

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

The `VkApplicationInfo` structure is defined as:

```c
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 an extension-specific 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** is the version of the Vulkan API against which the application expects to run, encoded as described in **Version Numbers**. If `apiVersion` is `0` the implementation must ignore it, otherwise if the implementation does not support the requested `apiVersion`, or an effective...
substitute for `apiVersion`, it **must** return `VK_ERROR_INCOMPATIBLE_DRIVER`. 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`.

### Valid Usage

- If `apiVersion` is not `0`, then it **must** be greater or equal to `VK_API_VERSION_1_0`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_APPLICATION_INFO`
- `pNext` **must** be `NULL`
- If `pApplicationName` is not `NULL`, `pApplicationName` **must** be a null-terminated UTF-8 string
- If `pEngineName` is not `NULL`, `pEngineName` **must** be a null-terminated UTF-8 string

To destroy an instance, call:

```c
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

- All child objects created using `instance` **must** have been destroyed prior to destroying `instance`
- If `VkAllocationCallbacks` were provided when `instance` was created, a compatible set of callbacks **must** be provided here
- If no `VkAllocationCallbacks` were provided when `instance` was created, `pAllocator` **must** be `NULL`

### Valid Usage (Implicit)

- If `instance` is not `NULL`, `instance` **must** be a valid `VkInstance` handle
- 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 4. 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
VK_DEFINE_HANDLE(VkPhysicalDevice)
```

### 4.1. Physical Devices

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

```c
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, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available physical devices were returned.
Valid Usage (Implicit)

- instance must be a valid VkInstance handle
- pPhysicalDeviceCount must be a valid pointer to a uint32_t value
- 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
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)

- physicalDevice must be a valid VkPhysicalDevice handle
- 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.

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:

```cpp
typedef enum VkVendorId {
    VK_VENDOR_ID_VIV = 0x10001,
    VK_VENDOR_ID_VSI = 0x10002,
    VK_VENDOR_ID_KAZAN = 0x10003,
    VK_VENDOR_ID_CODEPLAY = 0x10004,
} VkVendorId;
```

Note
Khronos vendor IDs may be allocated by vendors at any time. 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 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 may be returned in VkPhysicalDeviceProperties::deviceType are:

```cpp
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;
```
• **VK_PHYSICAL_DEVICE_TYPE_OTHER** - the device does not match any other available types.

• **VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU** - the device is typically one embedded in or tightly coupled with the host.

• **VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU** - the device is typically a separate processor connected to the host via an interlink.

• **VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU** - the device is typically a virtual node in a virtualization environment.

• **VK_PHYSICAL_DEVICE_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 may correlate with other advertised properties or capabilities of the system, such as how many memory heaps there are.

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

```c
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)

- **physicalDevice** must be a valid `VkPhysicalDevice` handle

- **pQueueFamilyPropertyCount** must be a valid pointer to a `uint32_t` value

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

The `VkQueueFamilyProperties` structure is defined as:
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 `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 `VkOffset3D` parameter must be integer multiples of A_x, A_y, and A_z, respectively.
  - width of a `VkExtent3D` parameter must be an integer multiple of A_x, or else x + width must equal the width of the image subresource corresponding to the parameter.
  - height of a `VkExtent3D` parameter must be an integer multiple of A_y, or else y + height must equal the height of the image subresource corresponding to the parameter.
  - depth of a `VkExtent3D` parameter must be an integer multiple of A_z, or else z + depth 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 must be scaled up by the compressed texel block dimensions.

Queues supporting graphics and/or compute operations must report (1,1,1) in
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 required to support whole mip level transfers, thus minImageTransferGranularity for queues belonging to such queue families may be (0,0,0).

The Device Memory section describes memory properties queried from the physical device.

For physical device feature queries see the Features chapter.

Bits which may be set in VkQueueFamilyProperties::queueFlags indicating capabilities of queues in a queue family are:

```c
typedef enum VQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,
    VK_QUEUE_COMPUTE_BIT = 0x00000002,
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
} VQueueFlagBits;
```

- **VK_QUEUE_GRAPHICS_BIT** 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.

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.

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
typedef VkFlags VQueueFlags;
```

VQueueFlags is a bitmask type for setting a mask of zero or more VQueueFlagBits.
4.2. 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. An application must create a separate logical device for each physical device it will use. 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.

4.2.1. Device Creation

Logical devices are represented by VkDevice handles:

```c
VK_DEFINE_HANDLE(VkDevice)
```

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

```c
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 `pEnabledExtensionNames` 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. 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.

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

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

### Valid Usage (Implicit)

- `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `pCreateInfo` must be a valid pointer to a valid `VkDeviceCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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:
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 an extension-specific 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**

- The **queueFamilyIndex** member of each element of **pQueueCreateInfos** must be unique within **pQueueCreateInfos**.
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- `pQueueCreateInfos` must be a valid pointer to an array of `queueCreateInfoCount` valid `VkDeviceQueueCreateInfo` structures
- If `enabledLayerCount` is not `0`, `ppEnabledLayerNames` must be a valid pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings
- If `enabledExtensionCount` is not `0`, `ppEnabledExtensionNames` must be a valid pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings
- If `pEnabledFeatures` is not `NULL`, `pEnabledFeatures` must be a valid pointer to a valid `VkPhysicalDeviceFeatures` structure
- `queueCreateInfoCount` must be greater than `0`

```
typedef VkFlags VkDeviceCreateFlags;
```

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

### 4.2.2. Device Use

The following is a high-level list of `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.

### 4.2.3. 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 run-time 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` 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`. 
4.2.4. Device Destruction

To destroy a device, call:

```c
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**

- All child objects created on `device` must have been destroyed prior to destroying `device`
- If `VkAllocationCallbacks` were provided when `device` was created, a compatible set of callbacks must be provided here
- If no `VkAllocationCallbacks` were provided when `device` was created, `pAllocator` must be `NULL`

---

**Valid Usage (Implicit)**

- If `device` is not `NULL`, `device` must be a valid `VkDevice` handle
- 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
4.3. Queues

4.3.1. 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.

Note

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.

4.3.2. 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
VK_DEFINE_HANDLE(VkQueue)
```

The `VkDeviceQueueCreateInfo` structure is defined as:

```c
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 an extension-specific structure.
- `flags` is reserved for future use.
- `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

- `queueFamilyIndex` **must** be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties`
- `queueCount` **must** be less than or equal to the `queueCount` member of the `VkQueueFamilyProperties` structure, as returned by `vkGetPhysicalDeviceQueueFamilyProperties` in the `pQueueFamilyProperties[queueFamilyIndex]`
- Each element of `pQueuePriorities` **must** be between 0.0 and 1.0 inclusive

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be 0
- `pQueuePriorities` **must** be a valid pointer to an array of `queueCount float` values
- `queueCount` **must** be greater than 0

```c
typedef VkFlags VkDeviceQueueCreateFlags;
```

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

To retrieve a handle to a `VkQueue` object, call:
```c
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.

### Valid Usage

- `queueFamilyIndex must` be one of the queue family indices specified when `device` was created, via the `VkDeviceQueueCreateInfo` structure
- `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
- `VkDeviceQueueCreateInfo::flags must` have been set to zero when `device` was created

### Valid Usage (Implicit)

- `device must` be a valid `VkDevice` handle
- `pQueue must` be a valid pointer to a `VkQueue` handle

#### 4.3.3. 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.

### 4.3.4. 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 make 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.

### 4.3.5. Queue Submission

Work is submitted to a queue via queue submission commands such as 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.

4.3.6. Queue Destruction

Queues are created along with a logical device during \texttt{vkCreateDevice}. All queues associated with a logical device are destroyed when \texttt{vkDestroyDevice} is called on that device.
Chapter 5. 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
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.

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.

5.1. 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, \texttt{vkCmd*} commands \textbf{can} be used to record to the command buffer.

**Executable**

\texttt{vkEndCommandBuffer} ends the recording of a command buffer, and moves it from the recording state to the \textit{executable state}. Executable command buffers \textbf{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 \textit{pending state}. Whilst in the pending state, applications \textbf{must} not attempt to modify the command buffer in any way - as the device \textbf{may} be processing the commands recorded to it. Once execution of a command buffer completes, the command buffer either reverts back to the \textit{executable state}, or if it was recorded with \texttt{VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT}, it moves to the \textit{invalid state}. A synchronization command \textbf{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 \textit{invalid state}. Command buffers in the invalid state \textbf{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 \textbf{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 \textit{initial state}. Resetting occurs as a result of \texttt{vkResetCommandBuffer} or \texttt{vkResetCommandPool}, or as part of \texttt{vkBeginCommandBuffer} (which additionally puts the command buffer in the \textit{recording state}).

Secondary command buffers \textbf{can} be recorded to a primary command buffer via \texttt{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 \textit{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.

5.2. 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:

```cpp
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkCommandPool)
```

To create a command pool, call:

```c
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

- pCreateInfo->queueFamilyIndex must be the index of a queue family available in the logical device device.
**Valid Usage (Implicit)**

- **device** must be a valid `VkDevice` handle
- **pCreateInfo** must be a valid pointer to a valid `VkCommandPoolCreateInfo` structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **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
typedef struct VkCommandPoolCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkCommandPoolCreateFlags     flags;
    uint32_t                      queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to an extension-specific structure.
- **flags** is a bitmask of `VkCommandPoolCreateFlagBits` indicating usage behavior for the pool and command buffers allocated from it.
- **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**
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO`
- `pNext` must be `NULL`
- `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:

```cpp
typedef enum VkCommandPoolCreateFlagBits {
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001,
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002,
} 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.

```cpp
typedef VkFlags VkCommandPoolCreateFlags;
```

`VkCommandPoolCreateFlags` is a bitmask type for setting a mask of zero or more `VkCommandPoolCreateFlagBits`.

To reset a command pool, call:

```cpp
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**

- All VkCommandBuffer objects allocated from commandPool must not be in the pending state

**Valid Usage (Implicit)**

- device must be a valid VkDevice handle
- commandPool must be a valid VkCommandPool handle
- flags must be a valid combination of VkCommandPoolResetFlagBits values
- 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_HOST_MEMORY
- VK_ERROR_OUT_OF_DEVICE_MEMORY

Bits which can be set in vkResetCommandPool::flags to control the reset operation are:

```c
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.

```c
typedef VkFlags VkCommandPoolResetFlags;
```

VkCommandPoolResetFlags is a bitmask type for setting a mask of zero or more VkCommandPoolResetFlagBits.
To destroy a command pool, call:

```c
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**
- All `VkCommandBuffer` objects allocated from `commandPool` must not be in the pending state
- If `VkAllocationCallbacks` were provided when `commandPool` was created, a compatible set of callbacks must be provided here
- If no `VkAllocationCallbacks` were provided when `commandPool` was created, `pAllocator` must be NULL

**Valid Usage (Implicit)**
- `device` must be a valid `VkDevice` handle
- If `commandPool` is not `VK_NULL_HANDLE`, `commandPool` must be a valid `VkCommandPool` handle
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `commandPool` is a valid handle, it must have been created, allocated, or retrieved from `device`

**Host Synchronization**
- Host access to `commandPool` must be externally synchronized
To allocate command buffers, call:

```c
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.

When command buffers are first allocated, they are in the initial state.

### Valid Usage (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pAllocateInfo` **must** be a valid pointer to a valid `VkCommandBufferAllocateInfo` structure
- `pCommandBuffers` **must** be a valid pointer to an array of `pAllocateInfo->commandBufferCount` `VkCommandBuffer` handles
- The value referenced by `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:
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 an extension-specific 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

• **commandBufferCount** **must** be greater than 0

### Valid Usage (Implicit)

• **sType** **must** be VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO
• **pNext** **must** be NULL
• **commandPool** **must** be a valid VkCommandPool handle
• **level** **must** be a valid VkCommandBufferLevel value

Possible values of VkCommandBufferAllocateInfo::level, specifying the command buffer level, are:

```
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:

```
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

- **commandBuffer** must not be in the pending state
- **commandBuffer** must have been allocated from a pool that was created with the **VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT**

### Valid Usage (Implicit)

- **commandBuffer** must be a valid **VkCommandBuffer** handle
- **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_HOST_MEMORY**
- **VK_ERROR_OUT_OF_DEVICE_MEMORY**

Bits which can be set in **vkResetCommandBuffer::flags** to control the reset operation are:

```c
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.
typedef VkFlags VkCommandBufferResetFlags;

VkCommandBufferResetFlags is a bitmask type for setting a mask of zero or more VkCommandBufferResetFlagBits.

To free command buffers, call:

```c
void vkFreeCommandBuffers(
    VkDevice device,  // device is the logical device that owns the command pool.
    VkCommandPool commandPool,  // commandPool is the command pool from which the command buffers were allocated.
    uint32_t commandBufferCount,  // commandBufferCount is the length of the pCommandBuffers array.
    const VkCommandBuffer* pCommandBuffers);  // 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**

- All elements of pCommandBuffers must not be in the pending state
- 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)**

- device must be a valid VkDevice handle
- commandPool must be a valid VkCommandPool handle
- commandBufferCount must be greater than 0
- commandPool must have been created, allocated, or retrieved from device
- 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
5.4. Command Buffer Recording

To begin recording a command buffer, call:

```c
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

- `commandBuffer` must not be in the recording or pending state.
- 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.
- If `commandBuffer` is a secondary command buffer, the `pInheritanceInfo` member of `pBeginInfo` must be a valid `VkCommandBufferInheritanceInfo` structure.
- 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`.
- 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)

- `commandBuffer` must be a valid `VkCommandBuffer` handle.
- `pBeginInfo` must be a valid pointer to a valid `VkCommandBufferBeginInfo` structure.

### 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

The VkCommandBufferBeginInfo structure is defined as:

```c
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 an extension-specific 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

- If `flags` contains `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`, the `renderPass` member of `pInheritanceInfo` must be a valid `VkRenderPass`
- 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`
- 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)

• **sType** must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO`
• **pNext** must be `NULL`
• **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
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
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:

```c
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 an extension-specific 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

- If the **inherited queries** feature is not enabled, `occlusionQueryEnable` must be `VK_FALSE`
- If the **inherited queries** feature is enabled, `queryFlags` must be a valid combination of `VkQueryControlFlagBits` values
- If the **inherited queries** feature is not enabled, `queryFlags` must be `0`
- If the **pipeline statistics queries** feature is enabled, `pipelineStatistics` must be a valid combination of `VkQueryPipelineStatisticFlagBits` values
- If the **pipeline statistics queries** feature is not enabled, `pipelineStatistics` must be `0`

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO`
- `pNext` must be `NULL`
- 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**.

Once recording starts, an application records a sequence of commands (`vkCmd*`) to set state in the command buffer, draw, dispatch, and other commands.

To complete recording of a command buffer, call:

```c
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

- \textit{commandBuffer} must be in the \textit{recording state}
- If \textit{commandBuffer} is a primary command buffer, there must not be an active render pass instance
- All queries made active during the recording of \textit{commandBuffer} must have been made inactive

Valid Usage (Implicit)

- \textit{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle

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

Return Codes

**Success**
- \texttt{VK_SUCCESS}

**Failure**
- \texttt{VK_ERROR_OUT_OF_HOST_MEMORY}
- \texttt{VK_ERROR_OUT_OF_DEVICE_MEMORY}

When a command buffer is in the executable state, it can be submitted to a queue for execution.

5.5. Command Buffer Submission

To submit command buffers to a queue, call:

\begin{verbatim}
VkResult vkQueueSubmit(
    VkQueue                                     queue,
    uint32_t                                    submitCount,
    const VkSubmitInfo*                         pSubmits,
    VkFence                                     fence);
\end{verbatim}

- \texttt{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**.

---

**Note**

Submission can be a high overhead operation, and applications **should** attempt to batch work together into as few calls to **vkQueueSubmit** as possible.

**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

• If fence is not VK_NULL_HANDLE, fence must be unsighaled

• 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

• 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

• 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

• Each element of the pSignalSemaphores member of any element of pSubmits must be unsighaled when the semaphore signal operation it defines is executed on the device

• 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

• All elements of the pWaitSemaphores member of all elements of pSubmits must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution

• Each element of the pCommandBuffers member of each element of pSubmits must be in the pending or executable state

• 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

• 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

• 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 pCommandBuffers member of each element of pSubmits must have been allocated from a VkCommandPool that was created for the same queue family queue belongs to

• If any element of 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

• Any resource created with VK_SHARING_MODE_EXCLUSIVE that is read by an operation specified by pSubmits must not be owned by any queue family other than the one which queue belongs to, at the time it is executed
Valid Usage (Implicit)

- queue must be a valid VkQueue handle
- If submitCount is not 0, pSubmits must be a valid pointer to an array of submitCount valid VkSubmitInfo structures
- If fence is not VK_NULL_HANDLE, fence must be a valid VkFence handle
- 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 VkSubmitInfo structure is defined as:
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 an extension-specific 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

- Each element of `pCommandBuffers` must not have been allocated with `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- If the geometry shaders feature is not enabled, each element of `pWaitDstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the tessellation shaders 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`
- Each element of `pWaitDstStageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SUBMIT_INFO`
- `pNext` must be `NULL`
- If `waitSemaphoreCount` is not 0, `pWaitSemaphores` must be a valid pointer to an array of `waitSemaphoreCount` valid `VkSemaphore` handles
- If `waitSemaphoreCount` is not 0, `pWaitDstStageMask` must be a valid pointer to an array of `waitSemaphoreCount` valid combinations of `VkPipelineStageFlagBits` values
- Each element of `pWaitDstStageMask` must not be 0
- If `commandBufferCount` is not 0, `pCommandBuffers` must be a valid pointer to an array of `commandBufferCount` valid `VkCommandBuffer` handles
- If `signalSemaphoreCount` is not 0, `pSignalSemaphores` must be a valid pointer to an array of `signalSemaphoreCount` valid `VkSemaphore` handles
- Each of the elements of `pCommandBuffers`, 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`

5.6. 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 there must be a prior signal of that semaphore that will not be consumed by a different wait on the semaphore.

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.
5.7. 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
void vkCmdExecuteCommands(
    VkCommandBuffer commandBuffer,         commandBuffer,
    uint32_t commandBufferCount,           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

- Each element of `pCommandBuffers` must have been allocated with a `level` of `VK_COMMAND_BUFFER_LEVEL_SECONDARY`
- Each element of `pCommandBuffers` must be in the pending or executable state
- 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
- 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`
- 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`
- 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
- 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`
- If `vkCmdExecuteCommands` is being called within a render pass instance, each element of `pCommandBuffers` must have been recorded with the `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`
- 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
- 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
- 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 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

**Valid Usage (Implicit)**

- commandBuffer must be a valid VkCommandBuffer handle
- pCommandBuffers must be a valid pointer to an array of commandBufferCount valid VkCommandBuffer handles
- commandBuffer must be in the recording state
- The VkCommandPool that commandBuffer was allocated from must support transfer, graphics, or compute operations
- commandBuffer must be a primary VkCommandBuffer
- commandBufferCount must be greater than 0
- 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</td>
<td>Compute</td>
</tr>
</tbody>
</table>
Chapter 6. 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.

6.1. 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.

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.

### 6.1.1. 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. If the old layout does not match the new layout, a transition occurs. The old layout must match the current layout of the image subresource range, with one exception. The old layout can always be specified as `VK_IMAGE_LAYOUT_UNDEFINED`, though doing so invalidates the
Note

Setting the old layout to `VK_IMAGE_LAYOUT_UNDEFINED` implies that the contents of the image subresource need not be preserved. Implementations may use this information to avoid performing expensive data transition operations.

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.

### 6.1.2. 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, but do execute `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`.

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 via the `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` and `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` pipeline stages.

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.

Bits which can be set, specifying pipeline stages, are:
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,
} VkPipelineStageFlagBits;

- **VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT** specifies the stage of the pipeline where any commands are initially received by the queue.
- **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT** specifies the stage of the pipeline where Draw/DispatchIndirect data structures are consumed.
- **VK_PIPELINE_STAGE_VERTEX_INPUT_BIT** specifies the stage of the pipeline where vertex and index buffers are consumed.
- **VK_PIPELINE_STAGE_VERTEX_SHADER_BIT** specifies the vertex shader stage.
- **VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT** specifies the tessellation control shader stage.
- **VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT** specifies the tessellation evaluation shader stage.
- **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT** specifies the geometry shader stage.
- **VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT** specifies the fragment shader stage.
- **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.
- **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.
- **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 format.
• **VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT** specifies the execution of a compute shader.

• **VK_PIPELINE_STAGE_TRANSFER_BIT** specifies the following commands:
  ◦ All copy commands, including **vkCmdCopyQueryPoolResults**
  ◦ **vkCmdBlitImage**
  ◦ **vkCmdResolveImage**
  ◦ All clear commands, with the exception of **vkCmdClearAttachments**

• **VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT** specifies the final stage in the pipeline where operations generated by all commands complete execution.

• **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.

• **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_TOP_OF_PIPE_BIT**
  ◦ **VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT**
  ◦ **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_BOTTOM_OF_PIPE_BIT**

• **VK_PIPELINE_STAGE_ALL_COMMANDS_BIT** is equivalent to the logical OR of every other pipeline stage flag that is supported on the queue it is used with.
An execution dependency with only `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` in the destination stage mask will only prevent that stage from executing in subsequently submitted commands. As this stage does not perform any actual execution, this is not observable - in effect, it does not delay processing of subsequent commands. Similarly an execution dependency with only `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` in the source stage mask will effectively not wait for any prior commands to complete.

When defining a memory dependency, using only `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` or `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` would never make any accesses available and/or visible because these stages do not access memory.

`VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT` and `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` are useful for accomplishing layout transitions and queue ownership operations when the required execution dependency is satisfied by other means - for example, semaphore operations between queues.

```c
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.

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>VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT</td>
<td>None required</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_INPUT_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_VERTEX_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSellation_CONTROL_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_TESSellation_EVALUATION_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</td>
</tr>
<tr>
<td>VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT</td>
<td>VK_QUEUE_GRAPHICS_BIT</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>
</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.
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.

The order and set of pipeline stages executed by a given command is determined by the command’s pipeline type, as described below:

For the graphics pipeline, the following stages occur in this order:

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- 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_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT

For the compute pipeline, the following stages occur in this order:

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT

For the transfer pipeline, the following stages occur in this order:

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- `VK_PIPELINE_STAGE_TRANSFER_BIT`
- `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

For host operations, only one pipeline stage occurs, so no order is guaranteed:

- `VK_PIPELINE_STAGE_HOST_BIT`

### 6.1.3. 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. Each access type corresponds to a bit flag in `VkAccessFlagBits`.

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.

Access types that can be set in an access mask include:

```c
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,
} VkAccessFlagBits;
```

- `VK_ACCESS_INDIRECT_COMMAND_READ_BIT` specifies read access to indirect command data read as part of an indirect 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 `vkCmdBindVertexBuffers`. 
• **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, uniform texel buffer, storage texel buffer, sampled image, or storage image.

• **VK_ACCESS_SHADER_WRITE_BIT** specifies write access to a 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.

• **VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT** specifies write access to a color or 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.

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</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>
</tbody>
</table>

94 | Chapter 6. Synchronization and Cache Control
<table>
<thead>
<tr>
<th>Access flag</th>
<th>Supported pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_ACCESS_UNIFORM_READ_BIT</td>
<td>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_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_WRITE_BIT</td>
<td>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_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</td>
</tr>
<tr>
<td>VK_ACCESS_TRANSFER_WRITE_BIT</td>
<td>VK_PIPELINE_STAGE_TRANSFER_BIT</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>
</tbody>
</table>

If a memory object does not have the \texttt{VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT} property, then \texttt{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, \texttt{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 \texttt{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
The `vkQueueSubmit` command automatically performs 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.

```c
typedef VkFlags VkAccessFlags;
```

`VkAccessFlags` is a bitmask type for setting a mask of zero or more `VkAccessFlagBits`.

### 6.1.4. 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 set of sample (x, y, layer, sample) 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, it is a pixel granularity dependency. If N equals M, 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, it is a sample granularity dependency.
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, 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.

### 6.2. 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` 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 order in which command buffers are specified in the `pCommandBuffers` member of `VkSubmitInfo`, 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.

**Action and synchronization commands** recorded to a command buffer execute the `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` pipeline stage in **submission order** - forming an implicit execution dependency between this stage in each command.

**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.

6.3. 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.

Fences are represented by VkFence handles:

```c
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkFence)
```

To create a fence, call:

```c
VkResult vkCreateFence(
    VkDevice device,                     device,
    const VkFenceCreateInfo*            pCreateInfo,  pCreateInfo,
    const VkAllocationCallbacks*       pAllocator,   pAllocator,
    VkFence*                            pFence);     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)

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a valid pointer to a valid VkFenceCreateInfo structure
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid VkAllocationCallbacks structure
- `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:

```c
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 an extension-specific structure.
- `flags` is a bitmask of `VkFenceCreateFlagBits` specifying the initial state and behavior of the fence.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_FENCE_CREATE_INFO`
- `pNext` must be NULL
- `flags` must be a valid combination of `VkFenceCreateFlagBits` values

```c
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.

```c
typedef VkFlags VkFenceCreateFlags;
```

`VkFenceCreateFlags` is a bitmask type for setting a mask of zero or more `VkFenceCreateFlagBits`.

To destroy a fence, call:
**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**

- All **queue submission** commands that refer to **fence** **must** have completed execution
- If **VkAllocationCallbacks** were provided when **fence** was created, a compatible set of callbacks **must** be provided here
- If no **VkAllocationCallbacks** were provided when **fence** was created, **pAllocator** **must** be **NULL**

**Valid Usage (Implicit)**

- **device** **must** be a valid **VkDevice** handle
- If **fence** is not **VK_NULL_HANDLE**, **fence** **must** be a valid **VkFence** handle
- If **pAllocator** is not **NULL**, **pAllocator** **must** be a valid pointer to a valid **VkAllocationCallbacks** structure
- 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:

**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 5. Fence Object Status Codes

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_SUCCESS</td>
<td>The fence specified by fence is signaled.</td>
</tr>
<tr>
<td>VK_NOT_READY</td>
<td>The fence specified by fence 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. 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)

- device must be a valid VkDevice handle
- fence must be a valid VkFence handle
- 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
VkResult vkResetFences(
    VkDevice                                    device,
    uint32_t                                    fenceCount,
    const VkFence*                              pFences);
```

- 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.

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

- 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)

- device must be a valid VkDevice handle
- pFences must be a valid pointer to an array of fenceCount valid VkFence handles
- fenceCount must be greater than 0
- 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_HOST_MEMORY
- 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
VkResult vkWaitForFences(
    VkDevice device,
    uint32_t fenceCount,
    const VkFence* pFences,
    VkBool32 waitAll,
    uint64_t timeout);
```

- **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 up to timeout nanoseconds for the condition to become satisfied.

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 specified timeout period expires before the condition is satisfied, vkWaitForFences returns VK_TIMEOUT. If the condition is satisfied before timeout nanoseconds has expired, vkWaitForFences returns VK_SUCCESS.

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.
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)

- `device must` be a valid `VkDevice` handle
- `pFences must` be a valid pointer to an array of `fenceCount` valid `VkFence` handles
- `fenceCount must` be greater than 0
- 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.
6.4. Semaphores

Semaphores are a synchronization primitive that can be used to insert a dependency between queue operations. Semaphores have two states - signaled and unsignaled. 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.

Semaphores are represented by VkSemaphore handles:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSemaphore)
```

To create a semaphore, call:

```
VkResult vkCreateSemaphore(
    VkDevice device,                        // device
    const VkSemaphoreCreateInfo* pCreateInfo, // pCreateInfo
    const VkAllocationCallbacks* pAllocator, // pAllocator
    VkSemaphore* pSemaphore);               // 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.

This command creates a binary semaphore that has a boolean payload indicating whether the semaphore is currently signaled or unsignaled. When created, the semaphore is in the unsignaled state.

**Valid Usage (Implicit)**

- device must be a valid VkDevice handle
- pCreateInfo must be a valid pointer to a valid VkSemaphoreCreateInfo structure
- If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
- 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
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 an extension-specific structure.
• `flags` is reserved for future use.

Valid Usage (Implicit)
• `sType` must be VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO
• `pNext` must be NULL
• `flags` must be 0

```c
typedef VkFlags VkSemaphoreCreateFlags;
```

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

To destroy a semaphore, call:

```c
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

- All submitted batches that refer to semaphore must have completed execution
- If VkAllocationCallbacks were provided when semaphore was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when semaphore was created, pAllocator must be NULL

Valid Usage (Implicit)

- device must be a valid VkDevice handle
- If semaphore is not VK_NULL_HANDLE, semaphore must be a valid VkSemaphore handle
- If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
- 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

6.4.1. 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.

The first synchronization scope includes every command submitted in the same batch. Semaphore signal operations that are defined by vkQueueSubmit 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.

6.4.2. 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 to the unsignaled state.

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. Also, in the case of 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 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.

6.4.3. 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.

6.5. 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:
To create an event, call:

```c
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 (Implicit)

- `device` **must** be a valid `VkDevice` handle
- `pCreateInfo` **must** be a valid pointer to a valid `VkEventCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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_OFDEVICE_MEMORY`

The `VkEventCreateInfo` structure is defined as:
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 an extension-specific structure.
• **flags** is reserved for future use.

### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_EVENT_CREATE_INFO`
- **pNext** must be *NULL*
- **flags** must be 0

typedefVkFlags VkEventCreateFlags;

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

To destroy an event, call:

```c
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

- All submitted commands that refer to **event** must have completed execution
- If **VkAllocationCallbacks** were provided when **event** was created, a compatible set of callbacks must be provided here
- If no **VkAllocationCallbacks** were provided when **event** was created, **pAllocator** must be *NULL*
Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- If `event` is not `VK_NULL_HANDLE`, `event` must be a valid `VkEvent` handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- 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
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 (Implicit)

- **device** must be a valid `VkDevice` handle
- **event** must be a valid `VkEvent` handle
- **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
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 (Implicit)

- **device** must be a valid `VkDevice` handle
- **event** must be a valid `VkEvent` handle
- **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:

```
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

- `event` must not be waited on by a `vkCmdWaitEvents` command that is currently executing

Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle
- `event` must be a valid `VkEvent` handle
- `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

The state of an event can also be updated on the device by commands inserted in command buffers.

To set the state of an event to signaled from a device, call:

```c
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 when the event is signaled.

When `vkCmdSetEvent` is submitted to a queue, it defines an execution dependency on commands that were submitted before it, and defines an event signal operation which sets the event to the signaled state.

The first synchronization scope includes all commands that occur earlier in submission order. The synchronization scope is limited to operations on the pipeline stages determined by the source stage mask specified by `stageMask`.

The second synchronization scope includes only the event signal operation.

If `event` is already in the signaled state when `vkCmdSetEvent` is executed on the device, then `vkCmdSetEvent` has no effect, no event signal operation occurs, and no execution dependency is generated.
Valid Usage

- `stageMask` must not include `VK_PIPELINE_STAGE_HOST_BIT`

- If the geometry shaders feature is not enabled, `stageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`

- 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`

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle

- `event` must be a valid `VkEvent` handle

- `stageMask` must be a valid combination of `VkPipelineStageFlagBits` values

- `stageMask` must not be 0

- `commandBuffer` must be in the recording state

- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations

- This command must only be called outside of a render pass instance

- 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>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td>Compute</td>
</tr>
</tbody>
</table>

To set the state of an event to unsignaled from a device, call:
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.

When vkCmdResetEvent 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 on the pipeline stages determined by the source stage mask specified by stageMask.

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

- stageMask must not include VK_PIPELINE_STAGE_HOST_BIT
- If the geometry shaders feature is not enabled, stageMask must not contain VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- 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
- When this command executes, event must not be waited on by a vkCmdWaitEvents command that is currently executing
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `event` must be a valid `VkEvent` handle
- `stageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `stageMask` must not be 0
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations
- This command must only be called outside of a render pass instance
- 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>Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To wait for one or more events to enter the signaled state on a device, call:

```c
void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    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.
• 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.
• pBufferMemoryBarrierCount 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.

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 srcStageMask. Event signal operations performed by vkSetEvent are only included in the first synchronization scope if VK_PIPELINE_STAGE_HOST_BIT is included in 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 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.
Note

vkCmdWaitEvents is used with vkCmdSetEvent 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.

Unlike vkCmdPipelineBarrier, a queue family ownership transfer cannot be performed using vkCmdWaitEvents.

Note

Applications should be careful to avoid race conditions when using events. There is no direct ordering guarantee between a vkCmdResetEvent command and a vkCmdWaitEvents command submitted after it, so some other execution dependency must be included between these commands (e.g. a semaphore).
Valid Usage

- **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**.

- If the **geometry shaders** feature is not enabled, **srcStageMask** must not contain **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**.

- If the **geometry shaders** feature is not enabled, **dstStageMask** must not contain **VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT**.

- 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**.

- 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**.

- 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.

- Any pipeline stage included in **srcStageMask** or **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.

- Each element of **pMemoryBarriers**, **pBufferMemoryBarriers** or **pImageMemoryBarriers** must not have any access flag included in its **srcAccessMask** member if that bit is not supported by any of the pipeline stages in **srcStageMask**, as specified in the table of supported access types.

- Each element of **pMemoryBarriers**, **pBufferMemoryBarriers** or **pImageMemoryBarriers** must not have any access flag included in its **dstAccessMask** member if that bit is not supported by any of the pipeline stages in **dstStageMask**, as specified in the table of supported access types.

- The **srcQueueFamilyIndex** and **dstQueueFamilyIndex** members of any element of **pBufferMemoryBarriers** or **pImageMemoryBarriers** must be equal.

- 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.

- 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.

- The **srcAccessMask** member of each element of **pBufferMemoryBarriers** 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.

- The **dstAccessMask** member of each element of **pBufferMemoryBarriers** 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.
specified in the table of supported access types

- The `srcAccessMask` member of each element of `pImageMemoryBarriers` 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

- The `dstAccessMask` member of any element of `pImageMemoryBarriers` 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

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pEvents` must be a valid pointer to an array of `eventCount` valid `VkEvent` handles
- `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` must not be `0`
- `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` must not be `0`
- If `memoryBarrierCount` is not `0`, `pMemoryBarriers` must be a valid pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures
- If `bufferMemoryBarrierCount` is not `0`, `pBufferMemoryBarriers` must be a valid pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures
- If `imageMemoryBarrierCount` is not `0`, `pImageMemoryBarriers` must be a valid pointer to an array of `imageMemoryBarrierCount` valid `VkImageMemoryBarrier` structures
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations
- `eventCount` must be greater than `0`
- 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
6.6. Pipeline Barriers

`vkCmdPipelineBarrier` is a synchronization command that inserts a dependency between commands submitted to the same queue, or between commands in the same subpass.

To record a pipeline barrier, call:

```c
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 stage mask.
- **dstStageMask** is a bitmask of `VkPipelineStageFlagBits` specifying the destination stage mask.
- **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.

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

• If the geometry shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

• If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`.

• 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`.

• 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`.

• 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, and for which `srcStageMask` contains a subset of the bit values in `VkSubpassDependency::srcStageMask`, `dstStageMask` contains a subset of the bit values in `VkSubpassDependency::dstStageMask`, `dependencyFlags` is equal to `VkSubpassDependency::dependencyFlags`, `srcAccessMask` member of each element of `pMemoryBarriers` and `pImageMemoryBarriers` contains a subset of the bit values in `VkSubpassDependency::srcAccessMask`, and `dstAccessMask` member of each element of `pMemoryBarriers` and `pImageMemoryBarriers` contains a subset of the bit values in `VkSubpassDependency::dstAccessMask`.

• If `vkCmdPipelineBarrier` is called within a render pass instance, `bufferMemoryBarrierCount` must be 0.

• If `vkCmdPipelineBarrier` is called within a render pass instance, the `image` member of any element of `pImageMemoryBarriers` must be equal to one of the elements of `pAttachments` that the current framebuffer was created with, that is also referred to by one of the elements of the `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment` members of the `VkSubpassDescription` instance that the current subpass was created with.

• If `vkCmdPipelineBarrier` is called within a render pass instance, the `oldLayout` and `newLayout` members of any element of `pImageMemoryBarriers` must be equal to the layout member of an element of the `pColorAttachments`, `pResolveAttachments` or `pDepthStencilAttachment` members of the `VkSubpassDescription` instance that the current subpass was created with, that refers to the same `image`.

• If `vkCmdPipelineBarrier` is called within a render pass instance, the `oldLayout` and `newLayout` members of an element of `pImageMemoryBarriers` must be equal.

• If `vkCmdPipelineBarrier` is called within a render pass instance, the `srcQueueFamilyIndex` and `dstQueueFamilyIndex` members of any element of `pImageMemoryBarriers` must be `VK_QUEUE_FAMILY_IGNORED`.

• Any pipeline stage included in `srcStageMask` or `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`.

• 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 \texttt{srcStageMask}, as specified in the table of supported access types

- The \texttt{dstAccessMask} member of each element of \texttt{pMemoryBarriers} \textbf{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

- 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 \textbf{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

- 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 \textbf{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

- For any element of \texttt{pImageMemoryBarriers}, 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 \textbf{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

- For any element of \texttt{pImageMemoryBarriers}, 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 \textbf{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
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` must not be 0
- `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` must not be 0
- `dependencyFlags` must be a valid combination of `VkDependencyFlagBits` values
- If `memoryBarrierCount` is not 0, `pMemoryBarriers` must be a valid pointer to an array of `memoryBarrierCount` valid `VkMemoryBarrier` structures
- If `bufferMemoryBarrierCount` is not 0, `pBufferMemoryBarriers` must be a valid pointer to an array of `bufferMemoryBarrierCount` valid `VkBufferMemoryBarrier` structures
- If `imageMemoryBarrierCount` is not 0, `pImageMemoryBarriers` must be a valid pointer to an array of `imageMemoryBarrierCount` valid `VkImageMemoryBarrier` structures
- `commandBuffer` must be in the recording state
- 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</td>
<td></td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Graphics</td>
<td>Compute</td>
</tr>
</tbody>
</table>

Bits which can be set in `vkCmdPipelineBarrier::dependencyFlags`, specifying how execution and memory dependencies are formed, are:

```c
typedef enum VkDependencyFlagBits {
    VK_DEPENDENCY_BY_REGION_BIT = 0x00000001,
} VkDependencyFlagBits;
```
• **VK_DEPENDENCY_BY_REGION_BIT** specifies that dependencies will be framebuffer-local.

```c
typedef VkFlags VkDependencyFlags;
```

**VkDependencyFlags** is a bitmask type for setting a mask of zero or more **VkDependencyFlagBits**.

### 6.6.1. Subpass Self-dependency

If `vkCmdPipelineBarrier` 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 `VkSubpassDependency` in the subpass dependency list for the render pass with `srcSubpass` and `dstSubpass` 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. Additionally, `srcStages` **must** not contain **VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT** in a self-dependency.

If the source and destination stage masks both include framebuffer-space stages, then `dependencyFlags` **must** include **VK_DEPENDENCY_BY_REGION_BIT**.

A `vkCmdPipelineBarrier` command inside a render pass instance **must** be a *subset* of one of the self-dependencies of the subpass it is used in, meaning that the stage masks and access masks **must** each include only a subset of the bits of the corresponding mask in that self-dependency. If the self-dependency has **VK_DEPENDENCY_BY_REGION_BIT** set, then so **must** the pipeline barrier. Pipeline barriers within a render pass instance **can** only be types `VkMemoryBarrier` or `VkImageMemoryBarrier`. If a `VkImageMemoryBarrier` is used, the image and image subresource range specified in the barrier **must** be a subset of one of the image views used by the framebuffer in the current subpass. Additionally, `oldLayout` **must** be equal to `newLayout`, and both the `srcQueueFamilyIndex` and `dstQueueFamilyIndex` **must** be **VK_QUEUE_FAMILY_IGNORED**.

### 6.7. 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.

#### 6.7.1. Global Memory Barriers

Global memory barriers apply to memory accesses involving all memory objects that exist at the time of its execution.

The **VkMemoryBarrier** structure is defined as:
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 an extension-specific 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`.

**Valid Usage (Implicit)**

- **sType** must be `VK_STRUCTURE_TYPE_MEMORY_BARRIER`
- **pNext** must be `NULL`
- **srcAccessMask** must be a valid combination of `VkAccessFlagBits` values
- **dstAccessMask** must be a valid combination of `VkAccessFlagBits` values

### 6.7.2. 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 `VkBufferMemoryBarrier` structure is defined as:

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 an extension-specific 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

- **offset must** be less than the size of **buffer**
- If **size** is not equal to **VK_WHOLE_SIZE**, **size must** be greater than 0
- If **size** is not equal to **VK_WHOLE_SIZE**, **size must** be less than or equal to than the size of **buffer** minus **offset**
- If **buffer** was created with a sharing mode of **VK_SHARING_MODE_CONCURRENT**, **srcQueueFamilyIndex** and **dstQueueFamilyIndex** must both be **VK_QUEUE_FAMILY_IGNORED**
- If **buffer** was created with a sharing mode of **VK_SHARING_MODE_EXCLUSIVE**, **srcQueueFamilyIndex** and **dstQueueFamilyIndex** must either both be **VK_QUEUE_FAMILY_IGNORED**, or both be a valid queue family (see **Queue Family Properties**)
- If **buffer** was created with a sharing mode of **VK_SHARING_MODE_EXCLUSIVE**, and **srcQueueFamilyIndex** and **dstQueueFamilyIndex** are not **VK_QUEUE_FAMILY_IGNORED**, at least one of them must be the same as the family of the queue that will execute this barrier
- If **buffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

Valid Usage (Implicit)

- **sType must** be **VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER**
- **pNext must** be **NULL**
- **buffer must** be a valid **VkBuffer** handle

6.7.3. 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 **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 an extension-specific 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 `oldLayout` is not equal to `newLayout`, then the memory barrier defines an image layout transition for the specified image subresource range.

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.
Valid Usage

- oldLayout must be VK_IMAGE_LAYOUT_UNDEFINED or the current layout of the image subresources affected by the barrier.

- newLayout must not be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED.

If image was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, srcQueueFamilyIndex and dstQueueFamilyIndex must both be VK_QUEUE_FAMILY_IGNORED.

If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, srcQueueFamilyIndex and dstQueueFamilyIndex must either both be VK_QUEUE_FAMILY_IGNORED, or both be a valid queue family (see Queue Family Properties).

If image was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and srcQueueFamilyIndex and dstQueueFamilyIndex are not VK_QUEUE_FAMILY_IGNORED, at least one of them must be the same as the family of the queue that will execute this barrier.

subresourceRange.baseMipLevel must be less than the mipLevels specified in VkImageCreateInfo when image was created.

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.

subresourceRange.baseArrayLayer must be less than the arrayLayers specified in VkImageCreateInfo when image was created.

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.

If image has a depth/stencil format with both depth and stencil components, then the aspectMask member of subresourceRange must include both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT.

If image has a color format, then the aspectMask member of subresourceRange must only include VK_IMAGE_ASPECT_COLOR_BIT.

If either oldLayout or newLayout is VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL then image must have been created with VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT set.

If either 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 set.

If either oldLayout or newLayout is VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL then image must have been created with VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT set.

If either oldLayout or newLayout is VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL then image must have been created with VK_IMAGE_USAGE_SAMPLED_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT set.

If either oldLayout or newLayout is VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL then image must have been created with VK_IMAGE_USAGE_TRANSFER_SRC_BIT set.

If either oldLayout or newLayout is VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL then image must have been created with VK_IMAGE_USAGE_TRANSFER_DST_BIT set.
• If image is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

Valid Usage (Implicit)

- sType must be VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER
-pNext must be NULL
- 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

6.7.4. 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. 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.

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.

**Note**

Whilst it is not invalid to provide destination or source access masks for memory barriers used for release or acquire operations, respectively, they have no practical effect. Access after a release operation has undefined results, and so visibility for those accesses has no practical effect. Similarly, write access before an acquire operation will produce undefined results for future access, so availability of those writes has no practical use. In an earlier version of the specification, these were required to match on both sides - but this was subsequently relaxed. These masks should be set to 0.

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`.

136 | Chapter 6. Synchronization and Cache Control
6.8. Wait Idle Operations

To wait on the host for the completion of outstanding queue operations for a given queue, call:

```
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)**

- `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:
vkDeviceWaitIdle(device);

- device is the logical device to idle.

vkDeviceWaitIdle is equivalent to calling vkQueueWaitIdle for all queues owned by device.

Valid Usage (Implicit)

- 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

6.9. Host Write Ordering Guarantees

When batches of command buffers are submitted to a queue via vkQueueSubmit, 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.
Chapter 7. 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:

```
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, 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, or input attachment for that subpass (as determined by the pColorAttachments, pDepthStencilAttachment, pResolveAttachments, 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.

**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
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.

### 7.1. Render Pass Creation

To create a render pass, call:

```c
VkResult vkCreateRenderPass(
    VkDevice device,               
    const VkRenderPassCreateInfo* pCreateInfo, 
    const VkAllocationCallbacks* pAllocator, 
    VkRenderPass* pRenderPass);
```

- `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)

- `device` must be a valid `VkDevice` handle
- `pCreateInfo` must be a valid pointer to a valid `VkRenderPassCreateInfo` structure
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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
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 an extension-specific 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` `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

- If the attachment member of any element of pInputAttachments, pColorAttachments, pResolveAttachments or pDepthStencilAttachment, or any element of pResolveAttachments or pDepthStencilAttachment in any element of pSubpasses is not VK_ATTACHMENT_UNUSED, it must be less than attachmentCount.

- 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.

- 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.

- 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.

- 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.

- The srcSubpass member of each element of pDependencies must be less than subpassCount.

- The dstSubpass member of each element of pDependencies must be less than subpassCount.

Valid Usage (Implicit)

- sType must be VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO.
- pNext must be NULL.
- flags must be 0.

- If attachmentCount is not 0, pAttachments must be a valid pointer to an array of attachmentCount valid VkAttachmentDescription structures.

- pSubpasses must be a valid pointer to an array of subpassCount valid VkSubpassDescription structures.

- If dependencyCount is not 0, pDependencies must be a valid pointer to an array of dependencyCount valid VkSubpassDependency structures.

- subpassCount must be greater than 0.

typedef VkFlags VkRenderPassCreateFlags;

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

The **VkAttachmentDescription** structure is defined as:

```c
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.

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

- `finalLayout **must** not be` `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`

- 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`

- If `format` is a depth/stencil format, `initialLayout **must** not be` `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`

- 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`

- If `format` is a depth/stencil format, `finalLayout **must** not be` `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_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
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
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:

```c
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 ...

146 | Chapter 7. Render Pass
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:

```c
typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
} 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`.

  **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.

The `VkSubpassDescription` structure is defined as:
```c
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 `pInputAttachments[X]`. Input attachments must also be bound to the pipeline in a descriptor set. If the `attachment` member of any element of `pInputAttachments` is `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, layer) framebuffer coordinates.

Each element of the `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 `pColorAttachments[X]`. If the `attachment` member of any element of `pColorAttachments` is `VK_ATTACHMENT_UNUSED`, writes to the corresponding location by a fragment are discarded.
If `pResolveAttachments` is not `NULL`, each of its elements corresponds to a color attachment (the element in `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 `VK_ATTACHMENT_UNUSED`.

If `pDepthStencilAttachment` is `NULL`, or if its attachment index is `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 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 that uses or preserves the attachment, and a subpass dependency from that subpass to this subpass.
- The attachment is not used or preserved in this subpass.

Once the contents of an attachment become undefined in subpass, they remain undefined for subpasses in subpass dependency chains starting with subpass until they are written again. However, they remain valid for subpasses in other subpass dependency chains starting with subpass if those subpasses use or preserve the attachment.
Valid Usage

- **pipelineBindPoint** must be `VK_PIPELINE_BIND_POINT_GRAPHICS`

- **colorAttachmentCount** must be less than or equal to `VkPhysicalDeviceLimits::maxColorAttachments`

- 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`

- 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`

- 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`

- If `pResolveAttachments` is not `NULL`, each resolve attachment that is not `VK_ATTACHMENT_UNUSED` must have the same `VkFormat` as its corresponding color attachment

- All attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` must have the same sample count

- All attachments in `pInputAttachments` that are not `VK_ATTACHMENT_UNUSED` must have formats whose features contain at least one of `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT` or `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`

- All attachments in `pColorAttachments` that are not `VK_ATTACHMENT_UNUSED` must have formats whose features contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`

- All attachments in `pResolveAttachments` that are not `VK_ATTACHMENT_UNUSED` must have formats whose features contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`

- If `pDepthStencilAttachment` is not `NULL` and the attachment is not `VK_ATTACHMENT_UNUSED` then it must have a format whose features contain `VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT`

- 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**
Valid Usage (Implicit)

- **flags** must be 0
- **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
typedef enum VkSubpassDescriptionFlagBits {
    } VkSubpassDescriptionFlagBits;
```

**Note**

All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

```c
typedef VkFlags VkSubpassDescriptionFlags;
```

`VkSubpassDescriptionFlags` is a bitmask type for setting a mask of zero or more `VkSubpassDescriptionFlagBits`.

The `VkAttachmentReference` structure is defined as:

```c
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

- 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_KHR, VK_IMAGE_LAYOUT_DEPTH_READ_ONLY_OPTIMAL_KHR, VK_IMAGE_LAYOUT_STENCIL_ATTACHMENT_OPTIMAL_KHR, or VK_IMAGE_LAYOUT_STENCIL_READ_ONLY_OPTIMAL_KHR

Valid Usage (Implicit)

- layout must be a valid VkImageLayout value

The VkSubpassDependency structure is defined as:

```c
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`. 

---

Chapter 7. Render Pass | 153
Valid Usage

- If the **geometry shaders** feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the **geometry shaders** feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- 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`
- 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`
- `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
- `srcSubpass` and `dstSubpass` must not both be equal to `VK_SUBPASS_EXTERNAL`
- 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`
- 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
- 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
- If `srcSubpass` equals `dstSubpass`, and `srcStageMask` and `dstStageMask` both include a framebuffer-space stage, then `dependencyFlags` must include `VK_DEPENDENCY_BY_REGION_BIT`

Valid Usage (Implicit)

- `srcStageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `srcStageMask` must not be `0`
- `dstStageMask` must be a valid combination of `VkPipelineStageFlagBits` values
- `dstStageMask` must not be `0`
- `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

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:
VkSubpassDependency implicitDependency = {
    .srcSubpass = VK_SUBPASS_EXTERNAL;
    .dstSubpass = firstSubpass; // First subpass attachment is used in
    .srcStageMask = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    .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_BOTTOM_OF_PIPE_BIT;
    .srcAccessMask = 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;
    .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.

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 \( \text{dstSubpass} \).

Automatic layout transitions away from \( \text{initialLayout} \) happens-after the availability operations for all dependencies with a \( \text{srcSubpass} \) equal to \( \text{VK_SUBPASS_EXTERNAL} \), where \( \text{dstSubpass} \) uses the attachment that will be transitioned. For attachments created with \( \text{VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT} \), automatic layout transitions away from \( \text{initialLayout} \) happen-after the availability operations for all dependencies with a \( \text{srcSubpass} \) equal to \( \text{VK_SUBPASS_EXTERNAL} \), where \( \text{dstSubpass} \) uses any aliased attachment.

Automatic layout transitions into \( \text{finalLayout} \) happens-before the visibility operations for all dependencies with a \( \text{dstSubpass} \) equal to \( \text{VK_SUBPASS_EXTERNAL} \), where \( \text{srcSubpass} \) uses the attachment that will be transitioned. For attachments created with \( \text{VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT} \), automatic layout transitions into \( \text{finalLayout} \) happen-before the visibility operations for all dependencies with a \( \text{dstSubpass} \) equal to \( \text{VK_SUBPASS_EXTERNAL} \), where \( \text{srcSubpass} \) uses any aliased attachment.

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 inputAttachmentAccess = 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 depthStencilAttachmentAccess =
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT;

VkSubpassDependency implicitDependency = {
    .srcSubpass = firstSubpass;
    .dstSubpass = secondSubpass;
    .srcStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .dstStageMask = inputAttachmentStages | depthStencilAttachmentStages;
    .srcAccessMask = inputAttachmentAccess | depthStencilAttachmentAccess;
    .dstAccessMask = inputAttachmentAccess | depthStencilAttachmentAccess;
    .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 `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.
- 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.

An attachment used as both an input attachment and a color attachment must be in the `VK_IMAGE_LAYOUT_GENERAL` layout. An attachment used as an input attachment and depth/stencil attachment must be in the `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.

To destroy a render pass, call:

```c
void vkDestroyRenderPass(
    VkDevice               device,  // device is the logical device that destroys the render pass.
    VkRenderPass           renderPass,  // renderPass is the handle of the render pass to destroy.
    const VkAllocationCallbacks* pAllocator);  // pAllocator controls host memory allocation as described in the Memory Allocation chapter.
```

**Valid Usage**

- All submitted commands that refer to `renderPass` must have completed execution.
- If `VkAllocationCallbacks` were provided when `renderPass` was created, a compatible set of callbacks must be provided here.
- If no `VkAllocationCallbacks` were provided when `renderPass` was created, `pAllocator` must be `NULL`.
Valid Usage (Implicit)

- **device** must be a valid VkDevice handle
- If **renderPass** is not VK_NULL_HANDLE, **renderPass** must be a valid VkRenderPass handle
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid VkAllocationCallbacks structure
- 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

7.2. 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.

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 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.

7.3. 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:
To create a framebuffer, call:

```c
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**

- 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)**

- `device` must be a valid `VkDevice` handle.
- `pCreateInfo` must be a valid pointer to a valid `VkFramebufferCreateInfo` structure.
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.
- `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
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 an extension-specific 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.
- `width`, `height` and `layers` define the dimensions of the framebuffer.

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.

**Note**

This restriction means 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`. 

160 | Chapter 7. Render Pass
Valid Usage

- **attachmentCount** must be equal to the attachment count specified in `renderPass`.
- 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.
- 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`.
- 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`.
- 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`.
- 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`.
- 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`.
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must have dimensions at least as large as the corresponding framebuffer dimension.
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must only specify a single mip level.
- If `flags` does not include `VK_FRAMEBUFFER_CREATE_IMAGELESS_BIT`, each element of `pAttachments` must have been created with the identity swizzle.
- **width** must be greater than 0.
- **width** must be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferWidth`.
- **height** must be greater than 0.
- **height** must be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferHeight`.
- **layers** must be greater than 0.
- **layers** must be less than or equal to `VkPhysicalDeviceLimits::maxFramebufferLayers`.
- 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.
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- `renderPass` must be a valid `VkRenderPass` handle
- 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`

Bits which can be set in `VkFramebufferCreateInfo::flags` to specify options for framebuffers are:

```c
typedef enum VkFramebufferCreateFlagBits {
} VkFramebufferCreateFlagBits;
```

**Note**
All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

```c
typedef VkFlags VkFramebufferCreateFlags;
```

`VkFramebufferCreateFlags` is a bitmask type for setting a mask of zero or more `VkFramebufferCreateFlagBits`.

To destroy a framebuffer, call:

```c
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

- All submitted commands that refer to `framebuffer` must have completed execution
- If `VkAllocationCallbacks` were provided when `framebuffer` was created, a compatible set of callbacks must be provided here
- If no `VkAllocationCallbacks` were provided when `framebuffer` was created, `pAllocator` must be `NULL`

### Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle
- If `framebuffer` is not `VK_NULL_HANDLE`, `framebuffer` must be a valid `VkFramebuffer` handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- If `framebuffer` is a valid handle, it must have been created, allocated, or retrieved from `device`

### Host Synchronization

- Host access to `framebuffer` must be externally synchronized

### 7.4. 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
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

- 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`.

- 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_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`.

- 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`.

- 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`.

- 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.
• 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)

• commandBuffer must be a valid VkCommandBuffer handle
• pRenderPassBegin must be a valid pointer to a valid VkRenderPassBeginInfo structure
• contents must be a valid VkSubpassContents 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
• 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 an extension-specific 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 must be contained within the framebuffer dimensions.

**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**
• **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**
• **renderPass** must be compatible with the **renderPass** member of the **VkFramebufferCreateInfo** structure specified when creating **framebuffer**
• **renderArea.offset.x** must be greater than or equal to 0
• **renderArea.offset.y** must be greater than or equal to 0
• **renderArea.offset.x + renderArea.offset.width** must be less than or equal to **VkFramebufferCreateInfo::width** the **framebuffer** was created with
• **renderArea.offset.y + renderArea.offset.height** must be less than or equal to **VkFramebufferCreateInfo::height** the **framebuffer** was created with
Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO`
- **pNext** must be `NULL`
- **renderPass** must be a valid `VkRenderPass` handle
- **framebuffer** must be a valid `VkFramebuffer` handle
- If `clearValueCount` is not 0, **pClearValues** must be a valid pointer to an array of `clearValueCount` `VkClearValue` unions
- Both of `framebuffer`, and **renderPass** must have been created, allocated, or retrieved from the same `VkDevice`

Possible values of `vkCmdBeginRenderPass::contents`, specifying how the commands in the first subpass will be provided, are:

```c
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`.

To query the render area granularity, call:

```c
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)

- **device** must be a valid `VkDevice` handle
- **renderPass** must be a valid `VkRenderPass` handle
- **pGranularity** must be a valid pointer to a `VkExtent2D` structure
- **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
void vkCmdNextSubpass(
    VkCommandBuffer commandBuffer,  // command buffer in which to record the command.  
    VkSubpassContents 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. 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

• The current subpass index **must** be less than the number of subpasses in the render pass minus one

Valid Usage (Implicit)

• **commandBuffer** **must** be a valid `VkCommandBuffer` handle
• **contents** **must** be a valid `VkSubpassContents` 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 inside of a render pass instance
• **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
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

- The current subpass index **must** be equal to the number of subpasses in the render pass minus one

Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `commandBuffer` **must** be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- `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>
Chapter 8. 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.

8.1. 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
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkShaderModule)
```

To create a shader module, call:

```c
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.

Valid Usage (Implicit)

• `device` must be a valid `VkDevice` handle

• `pCreateInfo` must be a valid pointer to a valid `VkShaderModuleCreateInfo` structure

• If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

• `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_OF_DEVICE_MEMORY`

The `VkShaderModuleCreateInfo` structure is defined as:

```c
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 an extension-specific 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

- `codeSize` must be greater than 0
- `codeSize` must be a multiple of 4
- `pCode` must point to valid SPIR-V code, formatted and packed as described by the Khronos SPIR-V Specification
- `pCode` must adhere to the validation rules described by the Validation Rules within a Module section of the SPIR-V Environment appendix
- `pCode` must declare the Shader capability for SPIR-V code
- `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
- If `pCode` declares any of the capabilities listed as optional in the SPIR-V Environment appendix, the corresponding feature(s) must be enabled

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- `pCode` must be a valid pointer to an array of $\frac{\text{codeSize}}{4}$ `uint32_t` values

```plaintext
typedef VkFlags VkShaderModuleCreateFlags;
```

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

To destroy a shader module, call:

```c
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

• If `VkAllocationCallbacks` were provided when `shaderModule` was created, a compatible set of callbacks must be provided here.

• If no `VkAllocationCallbacks` were provided when `shaderModule` was created, `pAllocator` must be `NULL`.

Valid Usage (Implicit)

• `device` must be a valid `VkDevice` handle.

• If `shaderModule` is not `VK_NULL_HANDLE`, `shaderModule` must be a valid `VkShaderModule` handle.

• If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.

• 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.

8.2. 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.

8.3. 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.

  **Note**
  
  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.

Stores issued to different memory locations within a single shader invocation **may** not be visible to other invocations, or **may** not become visible in the order they were performed.

The **OpMemoryBarrier** instruction **can** be used to provide stronger ordering of reads and writes performed by a single invocation. **OpMemoryBarrier** guarantees that any memory transactions issued by the shader invocation prior to the instruction complete prior to the memory transactions issued after the instruction. Memory barriers are needed for algorithms that require multiple invocations to access the same memory and require the operations to be performed in a partially-defined relative order. For example, if one shader invocation does a series of writes, followed by an **OpMemoryBarrier** instruction, followed by another write, then the results of the series of writes before the barrier become visible to other shader invocations at a time earlier or equal to when the results of the final write become visible to those invocations. In practice it means that another invocation that sees the results of the final write would also see the previous writes. Without the memory barrier, the final write **may** be visible before the previous writes.

Writes that are the result of shader stores through a variable decorated with **Coherent** automatically have available writes to the same buffer, buffer view, or image view made visible to them, and are themselves automatically made available to access by the same buffer, buffer view, or image view. Reads that are the result of shader loads through a variable decorated with **Coherent** automatically have available writes to the same buffer, buffer view, or image view made visible to them. The order that coherent writes to different locations become available is undefined, unless enforced by a memory barrier instruction or other memory dependency.

  **Note**
  
  Explicit memory dependencies **must** still be used to guarantee availability and visibility for access via other buffers, buffer views, or image views.

The built-in atomic memory transaction instructions **can** be used to read and write a given memory address atomically. While built-in atomic functions issued by multiple shader invocations are
executed in undefined order relative to each other, these functions perform both a read and a write of a memory address and guarantee that no other memory transaction will write to the underlying memory between the read and write. Atomic operations ensure automatic availability and visibility for writes and reads in the same way as those to Coherent variables.

**Note**

Memory accesses performed on different resource descriptors with the same memory backing may not be well-defined even with the Coherent decoration or via atomics, due to things such as image layouts or ownership of the resource - as described in the Synchronization and Cache Control chapter.

**Note**

Atomics allow shaders to use shared global addresses for mutual exclusion or as counters, among other uses.

### 8.4. 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.

### 8.5. 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 must include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.

#### 8.5.1. Vertex Shader Execution

A vertex shader must be executed at least once for each vertex specified by a draw command. 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`).

8.6. 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 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`.

8.6.1. Tessellation Control Shader Execution

A tessellation control shader is invoked at least once for each output vertex in a patch.

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.

8.7. 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.

8.7.1. Tessellation Evaluation Shader Execution

A tessellation evaluation shader is invoked at least once for each unique vertex generated by the tessellator.
8.8. 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.

8.8.1. 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.

8.9. 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.

8.9.1. 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.
  - Each covered sample must be included in at least one fragment shader invocation.
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.

Relative ordering of execution of different fragment shader invocations is explicitly not defined.

8.9.2. 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 early fragment tests are enabled, any depth value computed by the fragment shader has no effect.

8.10. 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.

8.11. 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.

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.

### 8.12. 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.

### 8.13. 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.
8.13.1. 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.

8.13.2. Device

All invocations executed on a single device form a 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.

8.13.3. 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 Device Scope, which can be used as a Memory Scope for barrier and atomic operations.

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.

8.13.4. 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.

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.

8.13.5. 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 \texttt{Flat} are uniform within a primitive scope instance.

### 8.13.6. Workgroup

A \textit{local workgroup} is a set of invocations that can synchronize and share data with each other using memory in the \texttt{Workgroup} storage class.

The \texttt{Workgroup Scope} can be used as both an \textit{Execution Scope} and \textit{Memory Scope} for barrier and atomic operations.

Each invocation in a local workgroup \textbf{must} be in the same \textit{command scope instance}.

Only compute shaders have defined workgroups - other shader types \textbf{cannot} use workgroup functionality. For shaders that have defined workgroups, this set of invocations forms an \textit{invocation group} as defined in the SPIR-V specification.

### 8.13.7. Quad

A \textit{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.

The specific set of invocations that make up a quad scope instance in other shader stages is undefined.

In a fragment shader, each invocation in a quad scope instance \textbf{must} be in the same \textit{primitive scope instance}.

For \textbf{shaders that have defined workgroups}, each invocation in a quad scope instance \textbf{must} be in the same \textit{local workgroup}.

In other shader stages, each invocation in a quad scope instance \textbf{must} be in the same \textit{device scope instance}.

Fragment shaders have defined quad scope instances.

### 8.13.8. Invocation

The smallest \textit{scope} is a single invocation; this is represented by the \texttt{Invocation Scope} in SPIR-V.

Fragment shader invocations \textbf{must} be in a \textit{primitive scope instance}.
Invocations in shaders that have defined workgroups **must** be in a local workgroup.

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.

### 8.14. 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 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 (\( \text{OpDPdx*} \), \( \text{OpDPdy*} \), and \( \text{OpFwidth*} \)) **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 that 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.

Derivative operations calculate their results as the difference between the result of \( P \) across invocations in the quad. For fine derivative operations (\( \text{OpDPdxFine} \) and \( \text{OpDPdyFine} \)), the values of \( \text{DPdx}(P_i) \) are calculated as

\[
\text{DPdx}(P_0) = \text{DPdx}(P_1) = P_1 - P_0
\]

\[
\text{DPdx}(P_2) = \text{DPdx}(P_3) = P_3 - P_2
\]

and the values of \( \text{DPdy}(P_j) \) are calculated as

\[
\text{DPdy}(P_0) = \text{DPdy}(P_2) = P_2 - P_0
\]

\[
\text{DPdy}(P_1) = \text{DPdy}(P_3) = P_3 - P_1
\]

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 **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 **should** use the fine derivative calculations described for \( P_0 \).
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:

\[
\begin{align*}
DPdx(P_n)' &= DPdx(P_n) / w \\
DPdy(P_n)' &= DPdy(P_n) / h
\end{align*}
\]

where \( w \) and \( h \) are the size of the fragments in the quad, and \( DPdx(P_n)' \) and \( DPdy(P_n)' \) are the pixel derivatives.

The results for \( \text{OpDPdx} \) and \( \text{OpDPdy} \) may be calculated as either fine or coarse derivatives, with implementations favouring the most efficient approach. Implementations 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 \( \text{OpImage}^*\text{Sample}^*\text{ImplicitLod} \) instruction is equivalent to executing \( \text{OpDPdx}(\text{Coordinate}) \) and \( \text{OpDPdy}(\text{Coordinate}) \), and passing the results as the \( \text{Grad} \) operands \( dx \) and \( dy \).

Note

It is expected that using the \( \text{ImplicitLod} \) variants of sampling functions will be substantially more efficient than using the \( \text{ExplicitLod} \) variants with explicitly generated derivatives.

### 8.15. Helper Invocations

When performing derivative 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 \( \text{HelperInvocation} \) built-in. Stores and atomics 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 operations.
Chapter 9. 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, or a compute pipeline.

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.

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, and graphics pipelines are each represented by `VkPipeline` handles:

```c
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipeline)
```

### 9.1. 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:
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**

- 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

- 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
Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` must be a valid `VkPipelineCache` handle
- `pCreateInfos` must be a valid pointer to an array of `CreateInfoCount` valid `VkComputePipelineCreateInfo` structures
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `pPipelines` must be a valid pointer to an array of `CreateInfoCount` `VkPipeline` handles
- `CreateInfoCount` must be greater than 0
- If `pipelineCache` is a valid handle, it 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`

The `VkComputePipelineCreateInfo` structure is defined as:

```c
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 an extension-specific 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

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` **must** be a valid handle to a compute `VkPipeline`
- 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
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not -1, `basePipelineHandle` **must** be `VK_NULL_HANDLE
- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` **must** be -1
- The `stage` member of `stage` **must** be `VK_SHADER_STAGE_COMPUTE_BIT`
- 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
- `layout` **must** be consistent with the layout of the compute shader specified in `stage`
- The number of resources in `layout` accessible to the compute shader stage **must** be less than or equal to `VkPhysicalDeviceLimits::maxPerStageResources`

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be a valid combination of `VkPipelineCreateFlagBits` values
- `stage` **must** be a valid `VkPipelineShaderStageCreateInfo` structure
- `layout` **must** be a valid `VkPipelineLayout` handle
- 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:
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 an extension-specific 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**.

Chapter 9. Pipelines | 191
Valid Usage

- If the geometry shaders feature is not enabled, stage must not be `VK_SHADER_STAGE_GEOMETRY_BIT`.
- 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`.
- Stage must not be `VK_SHADER_STAGE_ALL_GRAPHICS`, or `VK_SHADER_STAGE_ALL`.
- pName must be the name of an OpEntryPoint in module with an execution model that matches stage.
- 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`.
- 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`.
- 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`.
- 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`.
- 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.
- 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

Valid Usage (Implicit)

• sType must be VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO
• pNext must be NULL
• flags must be 0
• 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

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:

typedef enum VkPipelineShaderStageCreateFlagBits {
} VkPipelineShaderStageCreateFlagBits;

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,
} 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.

**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.

```typedef VkFlags VkShaderStageFlags;
```

**VkShaderStageFlags** is a bitmask type for setting a mask of zero or more **VkShaderStageFlagBits**.

### 9.2. Graphics Pipelines

Graphics pipelines consist of multiple shader stages, multiple fixed-function pipeline stages, and a pipeline layout.

To create graphics pipelines, call:

```c
VkResult vkCreateGraphicsPipelines(
    VkDevice device, 
    VkPipelineCache pipelineCache, 
    uint32_t createInfoCount, 
    const VkGraphicsPipelineCreateInfo* pCreateInfos, 
    const VkAllocationCallbacks* pAllocator, 
    VkPipeline* pPipelines);
```

- **device** is the logical device that creates the graphics 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 **VkGraphicsPipelineCreateInfo** 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 graphics pipeline objects are returned.
The 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**

- 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.
- 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.

**Valid Usage (Implicit)**

- device must be a valid VkDevice handle.
- If pipelineCache is not VK_NULL_HANDLE, pipelineCache must be a valid VkPipelineCache handle.
- pCreateInfos must be a valid pointer to an array of createInfoCount valid VkGraphicsPipelineCreateInfo structures.
- If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure.
- pPipelines must be a valid pointer to an array of createInfoCount VkPipeline handles.
- createInfoCount must be greater than 0.
- If pipelineCache is a valid handle, it 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

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 VkPipelineDepthStencilStateCreateInfo* pDepthStencilState;
    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 an extension-specific 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.
- **pInputAssemblyState** is a pointer to a VkPipelineInputAssemblyStateCreateInfo structure which determines input assembly behavior, as described in Drawing Commands.
- **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
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.
Valid Usage

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is -1, `basePipelineHandle` must be a valid handle to a graphics `VkPipeline`.

- 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.

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineHandle` is not `VK_NULL_HANDLE`, `basePipelineIndex` must be `-1`.

- If `flags` contains the `VK_PIPELINE_CREATE_DERIVATIVE_BIT` flag, and `basePipelineIndex` is not `-1`, `basePipelineHandle` must be `VK_NULL_HANDLE`.

- The `stage` member of each element of `pStages` must be unique.

- The `stage` member of one element of `pStages` must be `VK_SHADER_STAGE_VERTEX_BIT`.

- The `stage` member of each element of `pStages` must not be `VK_SHADER_STAGE_COMPUTE_BIT`.

- If `pStages` includes a tessellation control shader stage, it must include a tessellation evaluation shader stage.

- If `pStages` includes a tessellation evaluation shader stage, it must include a tessellation control shader stage.

- 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.

- 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.

- 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.

- 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.

- 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.

- If `pStages` includes tessellation shader stages, the `topology` member of `pInputAssembly` must be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`.

- If the `topology` member of `pInputAssembly` is `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`, `pStages` must include tessellation shader stages.

- 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`.

- 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

- 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

- 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`

- 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

- 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` 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` 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 and the `subpass` uses color attachments, then for each color attachment in the `subpass` the `blendEnable` member of the corresponding element of the `pAttachment` member of `pNextBlendState` must be `VK_FALSE` if the attached image's `format` features does not contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT`

- If rasterization is not disabled and the `subpass` uses color attachments, the `attachmentCount` member of `pNextBlendState` must be equal to the `colorAttachmentCount` used to create `subpass`

- If no element of the `pDynamicStates` member of `pNextDynamicState` is `VK_DYNAMIC_STATE_VIEWPORT`, the `pViewports` member of `pNextViewportState` must be a valid pointer to an array of `pNextViewportState->viewportCount` valid `VkViewport` structures

- If no element of the `pDynamicStates` member of `pNextDynamicState` is `VK_DYNAMIC_STATE_SCISSOR`, the `pScissors` member of `pNextViewportState` must be a valid pointer to an array of `pNextViewportState->scissorCount` `VkRect2D` structures

- If the wide lines feature is not enabled, and no element of the `pDynamicStates` member of `pNextDynamicState` is `VK_DYNAMIC_STATE_LINE_WIDTH`, the `lineWidth` member of `pRasterizationState` must be 1.0

- If the `rasterizerDiscardEnable` member of `pNextRasterizationState` is `VK_FALSE`, `pNextViewportState` must be a valid pointer to a valid `VkPipelineViewportStateCreateInfo` structure

- If the `rasterizerDiscardEnable` member of `pNextRasterizationState` is `VK_FALSE`, `pNextMultisampleState` must be a valid pointer to a valid `VkPipelineMultisampleStateCreateInfo` structure

- If the `rasterizerDiscardEnable` member of `pNextRasterizationState` is `VK_FALSE`, and `subpass` uses a depth/stencil attachment, `pNextDepthStencilState` must be a valid pointer to a valid `VkPipelineDepthStencilStateCreateInfo` structure
• If the `rasterizerDiscardEnable` member of `pRasterizationState` is `VK_FALSE`, and `subpass` uses color attachments, `pColorBlendState` must be a valid pointer to a valid `VkPipelineColorBlendStateCreateInfo` structure.

• 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 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.

• `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 `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`.

• 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 `pStages` includes a vertex shader stage, `pVertexInputState` must be a valid pointer to a valid `VkPipelineVertexInputStateCreateInfo` structure.

• If `pStages` includes a vertex shader stage, `pInputAssemblyState` must be a valid pointer to a valid `VkPipelineInputAssemblyStateCreateInfo` structure.
Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO`
- **pNext** must be `NULL`
- **flags** must be a valid combination of `VkPipelineCreateFlagBits` values
- **pStages** must be a valid pointer to an array of `stageCount` valid `VkPipelineShaderStageCreateInfo` structures
- **pRasterizationState** must be a valid pointer to a valid `VkPipelineRasterizationStateCreateInfo` structure
- If **pDynamicState** is not `NULL`, **pDynamicState** must be a valid pointer to a valid `VkPipelineDynamicStateCreateInfo` structure
- **layout** must be a valid `VkPipelineLayout` handle
- **renderPass** must be a valid `VkRenderPass` handle
- **stageCount** must be greater than 0
- 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`, and `VkComputePipelineCreateInfo`, specifying how a pipeline is created, are:

```c
typedef enum VkPipelineCreateFlagBits {
    VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT = 0x00000001,
    VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT = 0x00000002,
    VK_PIPELINE_CREATE_DERIVATIVE_BIT = 0x00000004,
} 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.

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
typedef VkFlags VkPipelineCreateFlags;
```

`VkPipelineCreateFlags` is a bitmask type for setting a mask of zero or more `VkPipelineCreateFlagBits`.

Chapter 9. Pipelines | 201
The `VkPipelineDynamicStateCreateInfo` structure is defined as:

```c
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType                      sType;
    const void*                          pNext;
    VkPipelineDynamicStateCreateFlags    flags;
    uint32_t                             dynamicStateCount;
    const VkDynamicState*                pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific 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

- Each element of `pDynamicStates` **must** be unique

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO`
- `pNext` **must** be `NULL`
- `flags` **must** be `0`
- If `dynamicStateCount` is not `0`, `pDynamicStates` **must** be a valid pointer to an array of `dynamicStateCount` valid `VkDynamicState` values

```c
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:
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_STENCIL_COMPARE_MASK = 6,
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
    VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
} VkDynamicState;

- **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` 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`

### 9.2.1. Valid Combinations of Stages for Graphics Pipelines

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`
    ▪ `VkPipelineInputAssemblyStateCreateInfo`
### 9.3. Pipeline destruction

To destroy a graphics or compute pipeline, call:

```c
void vkDestroyPipeline(
    VkDevice device,            // Logical device
    VkPipeline pipeline,        // Handle of the 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

• All submitted commands that refer to pipeline must have completed execution

• If VkAllocationCallbacks were provided when pipeline was created, a compatible set of callbacks must be provided here

• If no VkAllocationCallbacks were provided when pipeline was created, pAllocator must be NULL

Valid Usage (Implicit)

• device must be a valid VkDevice handle

• If pipeline is not VK_NULL_HANDLE, pipeline must be a valid VkPipeline handle

• If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure

• 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

9.4. Multiple Pipeline Creation

Multiple pipelines can be created simultaneously by passing an array of VkGraphicsPipelineCreateInfo, or VkComputePipelineCreateInfo structures into the vkCreateGraphicsPipelines, 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.

9.5. 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.

### 9.6. 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:

```c
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineCache)
```

To create pipeline cache objects, call:

```c
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.
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` 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.

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)**

- `device` must be a valid `VkDevice` handle
- `pCreateInfo` must be a valid pointer to a valid `VkPipelineCacheCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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:
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 an extension-specific structure.
- `flags` is reserved for future use.
- `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

- If `initialDataSize` is not 0, it must be equal to the size of `pInitialData`, as returned by `vkGetPipelineCacheData` when `pInitialData` was originally retrieved.
- If `initialDataSize` is not 0, `pInitialData` must have been retrieved from a previous call to `vkGetPipelineCacheData`.

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO`
- `pNext` must be NULL
- `flags` must be 0
- If `initialDataSize` is not 0, `pInitialData` must be a valid pointer to an array of `initialDataSize` bytes

typedef VkFlags VkPipelineCacheCreateFlags;

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

Pipeline cache objects can be merged using the command:
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**

- **dstCache** must not appear in the list of source caches

**Valid Usage (Implicit)**

- **device** must be a valid **VkDevice** handle
- **dstCache** must be a valid **VkPipelineCache** handle
- **pSrcCaches** must be a valid pointer to an array of **srcCacheCount** valid **VkPipelineCache** handles
- **srcCacheCount** must be greater than 0
- **dstCache** must have been created, allocated, or retrieved from **device**
- 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
VkResult vkGetPipelineCacheData(
    VkDevice device,  
    VkPipelineCache pipelineCache,  
    size_t* pDataSize,  
    void* pData);
```

- `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 `vkGetPipelineCacheData` will return `VK_INCOMPLETE`. 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 7. 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)**

- **device** must be a valid `VkDevice` handle
- **pipelineCache** must be a valid `VkPipelineCache` handle
- **pDataSize** must be a valid pointer to a `size_t` value
- 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
- **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
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
void vkDestroyPipelineCache(
    VkDevice device,
    VkPipelineCache pipelineCache,
    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

• If `VkAllocationCallbacks` were provided when `pipelineCache` was created, a compatible set of callbacks must be provided here
• If no `VkAllocationCallbacks` were provided when `pipelineCache` was created, `pAllocator` must be NULL
Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle

- If `pipelineCache` is not `VK_NULL_HANDLE`, `pipelineCache` must be a valid `VkPipelineCache` handle

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

- 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

9.7. 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
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

- The offset member of each element of pMapEntries must be less than dataSize.
- The size member of each element of pMapEntries must be less than or equal to dataSize minus offset.

Valid Usage (Implicit)

- If mapEntryCount is not 0, pMapEntries must be a valid pointer to an array of mapEntryCount valid VkSpecializationMapEntry structures.
- If dataSize is not 0, pData must be a valid pointer to an array of dataSize bytes.

The VkSpecializationMapEntry structure is defined as:

```c
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

- 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:
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, 0 * sizeof(uint32_t), sizeof(uint32_t) },
    { 42, 1 * sizeof(uint32_t), sizeof(uint32_t) },
    { 3, 2 * sizeof(uint32_t), sizeof(uint32_t) }
};
const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info = {
    3, entries, // mapEntryCount, pMapEntries
    3 * sizeof(uint32_t), data, // dataSize, 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
%i32 = 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.

Now to specialize the above via the specialization constants mechanism:

```
struct SpecializationData {
    int32_t data0;
    float data1;
};

contVkSpecializationMapEntry entries[] =
{
    { 0, // constantID
        offsetof(SpecializationData, data0), // offset
        sizeof(SpecializationData::data0) // size
    },
    { 12, // constantID
        offsetof(SpecializationData, data1), // offset
        sizeof(SpecializationData::data1) // size
    }
};

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

contVkSpecializationInfo info =
{
    2, // mapEntryCount
    entries, // pMapEntries
    sizeof(data), // dataSize
    &data, // pData
};
```

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.

9.8. Pipeline Binding

Once a pipeline has been created, it can be bound to the command buffer using the command:

```c
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 whether to bind to the compute or graphics bind point. Binding one does not disturb the other.
- `pipeline` is the pipeline to be bound.

Once bound, a pipeline binding affects subsequent graphics or compute commands in the command buffer until a different pipeline is bound to the bind point. The pipeline bound to `VK_PIPELINE_BIND_POINT_COMPUTE` controls the behavior of `vkCmdDispatch` and `vkCmdDispatchIndirect`. The pipeline bound to `VK_PIPELINE_BIND_POINT_GRAPHICS` controls the behavior of all drawing commands. No other commands are affected by the pipeline state.

Valid Usage

- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, the `VkCommandPool` that `commandBuffer` was allocated from must support compute operations.
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations.
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_COMPUTE`, `pipeline` must be a compute pipeline.
- If `pipelineBindPoint` is `VK_PIPELINE_BIND_POINT_GRAPHICS`, `pipeline` must be a graphics pipeline.
- 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.
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value
- `pipeline` must be a valid `VkPipeline` handle
- `commandBuffer` must be in the `recording state`
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations
- 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

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>

Possible values of `vkCmdBindPipeline::pipelineBindPoint`, specifying the bind point of a pipeline object, are:

```cpp
typedef enum VkPipelineBindPoint {
    VK_PIPELINE_BIND_POINT_GRAPHICS = 0,
    VK_PIPELINE_BIND_POINT_COMPUTE = 1,
} VkPipelineBindPoint;
```

- `VK_PIPELINE_BIND_POINT_COMPUTE` specifies binding as a compute pipeline.
- `VK_PIPELINE_BIND_POINT_GRAPHICS` specifies binding as a graphics pipeline.

9.9. 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.
Chapter 10. Memory Allocation

Vulkan memory is broken up into two categories, host memory and device memory.

10.1. 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
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

- **pfnAllocation** must be a valid pointer to a valid user-defined `PFN_vkAllocationFunction`.
- **pfnReallocation** must be a valid pointer to a valid user-defined `PFN_vkReallocationFunction`.
- **pfnFree** must be a valid pointer to a valid user-defined `PFN_vkFreeFunction`.
- If either of **pfnInternalAllocation** or **pfnInternalFree** is not NULL, both must be valid callbacks.

The type of **pfnAllocation** is:

```c
typedef void* (VKAPI_PTR *PFN_vkAllocationFunction)(
    void* pUserData,
    size_t size,
    size_t alignment,
    VkSystemAllocationScope allocationScope);
```

- **pUserData** is the value specified for `VkAllocationCallbacks::pUserData` in the allocator specified by the application.
- **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.

If **pfnAllocation** is unable to allocate the requested memory, it must return NULL. If the allocation was successful, it must return a valid pointer to memory allocation containing at least **size** bytes, and with the pointer value being a multiple of **alignment**.

**Note**

Correct Vulkan operation cannot be assumed if the application does not follow these rules.

For example, **pfnAllocation** (or **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 `VkInstance` objects are going to operate correctly (even `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 run-time 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 \texttt{PFN_vkReallocationFunction} is:

\begin{verbatim}
typedef void* (VKAPI_PTR *PFN_vkReallocationFunction)(
    void*                      pUserData,
    void*                      pOriginal,
    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{pOriginal} must be either NULL or a pointer previously returned by \texttt{PFN_vkReallocationFunction} or \texttt{PFN_vkAllocationFunction} of a compatible allocator.
- \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.

\texttt{PFN_vkReallocationFunction} must return an allocation with enough space for \texttt{size} bytes, and the contents of the original allocation from bytes zero to min(original size, new size) - 1 must be preserved in the returned allocation. If \texttt{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 \texttt{pOriginal} is NULL, then \texttt{PFN_vkReallocationFunction} must behave equivalently to a call to \texttt{PFN_vkAllocationFunction} with the same parameter values (without \texttt{pOriginal}).

If \texttt{size} is zero, then \texttt{PFN_vkReallocationFunction} must behave equivalently to a call to \texttt{PFN_vkFreeFunction} with the same \texttt{pUserData} parameter value, and \texttt{pMemory} equal to \texttt{pOriginal}.

If \texttt{pOriginal} is non-NULL, the implementation must ensure that \texttt{alignment} is equal to the \texttt{alignment} used to originally allocate \texttt{pOriginal}.

If this function fails and \texttt{pOriginal} is non-NULL the application must not free the old allocation.

\texttt{PFN_vkReallocationFunction} must follow the same rules for return values as \texttt{PFN_vkAllocationFunction}.

The type of \texttt{PFN_vkFreeFunction} is:

\begin{verbatim}
typedef void (VKAPI_PTR *PFN_vkFreeFunction)(
    void* pUserData,
    void* pMemory);
\end{verbatim}
• **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 **pfnInternalAllocation** is:

```c
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 **pfnInternalFree** is:

```c
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:
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 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 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
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 `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 `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.

10.2. 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
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)

• `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
• `pMemoryProperties` **must** be a valid pointer to a `VkPhysicalDeviceMemoryProperties` structure

The `VkPhysicalDeviceMemoryProperties` structure is defined as:

```c
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_MEMORY_PROPERTY_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_MEMORY_PROPERTY_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- `VK_MEMORY_PROPERTY_DEVICE_LOCAL_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_HOST_VISIBLE_BIT` | `VK_MEMORY_PROPERTY_HOST_CACHED_BIT` | `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` | `VK_MEMORY_PROPERTY_LAZY_ALLOCATED_BIT`

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.

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)

**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_MEMORY_PROPERTY_DEVICE_LOCAL_BIT was before VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, the list would still be in a valid order.

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`
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
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);

The VkMemoryHeap structure is defined as:

```c
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:

```cpp
typedef enum VkMemoryHeapFlagBits {
    VK_MEMORY_HEAP_DEVICE_LOCAL_BIT = 0x00000001,
} 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.

```cpp
typedef VkFlags VkMemoryHeapFlags;
```

`VkMemoryHeapFlags` is a bitmask type for setting a mask of zero or more `VkMemoryHeapFlagBits`.

The `VkMemoryType` structure is defined as:

```cpp
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:

```cpp
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,
} 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.

```plaintext
typedef VkFlags VkMemoryPropertyFlags;
```

**VkMemoryPropertyFlags** is a bitmask type for setting a mask of zero or more **VkMemoryPropertyFlagBits**.

A Vulkan device operates on data in device memory via memory objects that are represented in the API by a **VkDeviceMemory** handle:

```plaintext
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDeviceMemory)
```

To allocate memory objects, call:

```plaintext
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.
The maximum number of valid memory allocations that can exist simultaneously within a VkDevice may be restricted by implementation- or platform-dependent limits. If a call to vkAllocateMemory would cause the total number of allocations to exceed these limits, such a call will fail and must return VK_ERROR_TOO_MANY_OBJECTS. The maxMemoryAllocationCount feature describes the number of allocations that can exist simultaneously before encountering these internal limits.

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.

### Valid Usage

- **pAllocateInfo->allocationSize** must be less than or equal to VkPhysicalDeviceMemoryProperties::memoryHeaps[memindex].size where **memindex** = VkPhysicalDeviceMemoryProperties::memoryTypes[pAllocateInfo->memoryTypeIndex].heapIndex as returned by vkGetPhysicalDeviceMemoryProperties for the VkPhysicalDevice that device was created from

- **pAllocateInfo->memoryTypeIndex** must be less than VkPhysicalDeviceMemoryProperties::memoryTypeCount as returned by vkGetPhysicalDeviceMemoryProperties for the VkPhysicalDevice that device was created from

### Valid Usage (Implicit)

- **device** must be a valid VkDevice handle
- **pAllocateInfo** must be a valid pointer to a valid VkMemoryAllocateInfo structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid VkAllocationCallbacks structure
- **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_TOO_MANY_OBJECTS

The VkMemoryAllocateInfo structure is defined as:
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 an extension-specific 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

Valid Usage

• allocationSize must be greater than 0

Valid Usage (Implicit)

• sType must be VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO
• pNext must be NULL

To free a memory object, call:

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. If there are still any bound images or buffers, the memory may not be immediately released by the implementation, but must be released by the time all bound images and buffers have been destroyed. Once memory is 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

• All submitted commands that refer to memory (via images or buffers) must have completed execution

Valid Usage (Implicit)

• device must be a valid VkDevice handle
• If memory is not VK_NULL_HANDLE, memory must be a valid VkDeviceMemory handle
• If pAllocator is not NULL, pAllocator must be a valid pointer to a valid VkAllocationCallbacks structure
• If memory is a valid handle, it must have been created, allocated, or retrieved from device

Host Synchronization

• Host access to memory must be externally synchronized

10.2.1. Host Access to Device Memory Objects

Memory objects created with vkAllocateMemory are not directly host accessible.

Memory objects created with the memory property VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT are considered mappable. Memory objects 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:

```
VkResult vkMapMemory(
  VkDevice                                    device, 
  VkDeviceMemory                              memory, 
  VkDeviceSize                                offset, 
  VkDeviceSize                                size, 
  VkMemoryMapFlags                            flags, 
  void**                                      ppData);
```

• device is the logical device that owns the memory.
• **memory** is the **VkDeviceMemory** object to be mapped.

• **offset** is a zero-based byte offset from the beginning of the memory object.

• **size** is the size of the memory range to map, or **VK_WHOLE_SIZE** to map from **offset** to the end of the allocation.

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

• **pData** is a pointer to a **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 **VkPhysicalDeviceLimits**:minMemoryMapAlignment.

After a successful call to **vkMapMemory** the memory object **memory** is considered to be currently **host mapped**.

---

**Note**

It is an application error to call **vkMapMemory** on a memory object that is already **host mapped**.

---

**Note**

**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, **vkMapMemory** **must** return **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 **vkUnmapMemory**.

**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 **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 **VkPhysicalDeviceLimits**:nonCoherentAtomSize, and round the end of the range up to the nearest multiple of **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.

---

**Note**

It is important for the application developer to become meticulously familiar with all of the mechanisms described in the chapter on **Synchronization and Cache Control** as they are crucial to maintaining memory access ordering.
Valid Usage

- **memory** must not be currently host mapped
- **offset** must be less than the size of **memory**
- If **size** is not equal to `VK_WHOLE_SIZE`, **size** must be greater than 0
- If **size** is not equal to `VK_WHOLE_SIZE`, **size** must be less than or equal to the size of the **memory** minus **offset**
- **memory** must have been created with a memory type that reports `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`

Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- **memory** must be a valid `VkDeviceMemory` handle
- **flags** must be 0
- **pData** must be a valid pointer to a pointer value
- **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`

```cpp
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`. 
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.

To flush ranges of non-coherent memory from the host caches, call:

```c
VkResult vkFlushMappedMemoryRanges(
    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 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.

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)

- `device` must be a valid `VkDevice` handle
- `pMemoryRanges` must be a valid pointer to an array of `memoryRangeCount` valid `VkMappedMemoryRange` structures
- `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
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)

- **device** must be a valid `VkDevice` handle
- **pMemoryRanges** must be a valid pointer to an array of `memoryRangeCount` valid `VkMappedMemoryRange` structures
- **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
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 an extension-specific 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

- memory must be currently host mapped

- If size is not equal to VK_WHOLE_SIZE, offset and size must specify a range contained within the currently mapped range of memory

- If size is equal to VK_WHOLE_SIZE, offset must be within the currently mapped range of memory

- If size is equal to VK_WHOLE_SIZE, the end of the current mapping of memory must be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize bytes from the beginning of the memory object

- offset must be a multiple of VkPhysicalDeviceLimits::nonCoherentAtomSize

- 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)

- sType must be VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE

- pNext must be NULL

- 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
void vkUnmapMemory(
    VkDevice                                    device,
    VkDeviceMemory                              memory);
```

- device is the logical device that owns the memory.
- memory is the memory object to be unmapped.

Valid Usage

- memory must be currently host mapped

Valid Usage (Implicit)

- device must be a valid VkDevice handle

- memory must be a valid VkDeviceMemory handle

- memory must have been created, allocated, or retrieved from device
Host Synchronization

- Host access to memory must be externally synchronized

10.2.2. 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:

```c
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**

- memory must have been created with a memory type that reports VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT
Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- **memory** must be a valid `VkDeviceMemory` handle
- `pCommittedMemoryInBytes` must be a valid pointer to a `VkDeviceSize` value
- **memory** must have been created, allocated, or retrieved from `device`
Chapter 11. 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.

11.1. 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
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBuffer)
```

To create buffers, call:

```c
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**

- 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)

- **device** must be a valid `VkDevice` handle
- **pCreateInfo** must be a valid pointer to a valid `VkBufferCreateInfo` structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **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**

The `VkBufferCreateInfo` structure is defined as:

```c
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 an extension-specific 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` 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

- **size** must be greater than 0

  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `pQueueFamilyIndices` must be a valid pointer to an array of `queueFamilyIndexCount uint32_t` values

  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, `queueFamilyIndexCount` must be greater than 1

  If `sharingMode` is `VK_SHARING_MODE_CONCURRENT`, each element of `pQueueFamilyIndices` must be unique and must be less than `pQueueFamilyPropertyCount` returned by `vkGetPhysicalDeviceQueueFamilyProperties` for the `physicalDevice` that was used to create `device`

- If the `sparse bindings` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_BINDING_BIT`

- If the `sparse buffer residency` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT`

- If the `sparse aliased residency` feature is not enabled, `flags` must not contain `VK_BUFFER_CREATE_SPARSE_ALIASED_BIT`

- 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`

Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO`

- **pNext** must be `NULL`

- **flags** must be a valid combination of `VkBufferCreateFlagBits` values

- **usage** must be a valid combination of `VkBufferUsageFlagBits` values

- **usage** must not be 0

- **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,
} 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, or vkCmdDispatchIndirect.

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:
typedef enum VkBufferCreateFlagBits {
    VK_BUFFER_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_BUFFER_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
} 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.

See [Sparse Resource Features](#) and [Physical Device Features](#) for details of the sparse memory features supported on a device.

typedef VkFlags VkBufferCreateFlags;

**VkBufferCreateFlags** is a bitmask type for setting a mask of zero or more **VkBufferCreateFlagBits**.

To destroy a buffer, call:

```c
void vkDestroyBuffer(
    VkDevice device,              // device
    VkBuffer buffer,              // buffer
    const VkAllocationCallbacks* pAllocator); // 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.

### Valid Usage

- All submitted commands that refer to **buffer**, either directly or via a **VkBufferView**, must have completed execution
- If **VkAllocationCallbacks** were provided when **buffer** was created, a compatible set of callbacks must be provided here
- If no **VkAllocationCallbacks** were provided when **buffer** was created, **pAllocator** must be NULL
Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- If **buffer** is not `VK_NULL_HANDLE`, **buffer** must be a valid `VkBuffer` handle
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure
- If **buffer** is a valid handle, it must have been created, allocated, or retrieved from **device**

Host Synchronization

- Host access to **buffer** must be externally synchronized

11.2. 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:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkBufferView)
```

To create a buffer view, call:

```
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)

- **device** must be a valid `VkDevice` handle
- **pCreateInfo** must be a valid pointer to a valid `VkBufferViewCreateInfo` structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **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:

```c
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 an extension-specific 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

- **offset** must be less than the size of **buffer**
- **offset** must be a multiple of \( \text{VkPhysicalDeviceLimits::minTexelBufferOffsetAlignment} \)
- If **range** is not equal to **VK_WHOLE_SIZE**, **range** must be greater than 0
- If **range** is not equal to **VK_WHOLE_SIZE**, **range** must be an integer multiple of the texel block size of **format**
- If **range** is not equal to **VK_WHOLE_SIZE**, **range** divided by the texel block size of **format**, multiplied by the number of texels per texel block for that format (as defined in the Compatible Formats table), **must** be less than or equal to \( \text{VkPhysicalDeviceLimits::maxTexelBufferElements} \)
- 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**

**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**

- 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 in **VkFormatProperties::bufferFeatures** returned by **vkGetPhysicalDeviceFormatProperties**
- 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**
- If **buffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

Valid Usage (Implicit)

- **sType** **must** be **VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO**
- **pNext** **must** be **NULL**
- **flags** **must** be 0
- **buffer** **must** be a valid **VkBuffer** handle
- **format** **must** be a valid **VkFormat** value

```c
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
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**

- All submitted commands that refer to **bufferView** must have completed execution.
- If **VkAllocationCallbacks** were provided when **bufferView** was created, a compatible set of callbacks must be provided here.
- If no **VkAllocationCallbacks** were provided when **bufferView** was created, **pAllocator** must be **NULL**.

**Valid Usage (Implicit)**

- **device** must be a valid **VkDevice** handle.
- If **bufferView** is not **VK_NULL_HANDLE**, **bufferView** must be a valid **VkBufferView** handle.
- If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure.
- 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.

---

### 11.3. 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:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImage)
```
To create images, call:

```c
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**

- 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)**

- `device` must be a valid `VkDevice` handle
- `pCreateInfo` must be a valid pointer to a valid `VkImageCreateInfo` structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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
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 an extension-specific 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`

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.
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 `VkBool32 imageCreateMaybeLinear` indicate if the resultant image may be linear. (The definition below is trivial because certain extensions are disabled in this build of the specification).
  - If `tiling` is `VK_IMAGE_TILING_LINEAR`, then `imageCreateMaybeLinear` is true.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateMaybeLinear` is false.

- 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 `VkImageFormatProperties::linearTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.
  - If `tiling` is `VK_IMAGE_TILING_OPTIMAL`, then `imageCreateFormatFeatures` is value of `VkImageFormatProperties::optimalTilingFeatures` found by calling `vkGetPhysicalDeviceFormatProperties` with parameter `format` equal to `VkImageCreateInfo::format`.

- Let `uint32_t imageCreateMaxMipLevels` be the value of `VkImageFormatProperties::maxMipLevels` found by calling `vkGetPhysicalDeviceImageFormatProperties` with parameters `format`, `imageType`, `tiling`, `usage`, and `flags` equal to those in `VkImageCreateInfo`. If `vkGetPhysicalDeviceImageFormatProperties` returns an error, then `imageCreateMaxMipLevels` is undefined.

- Let `uint32_t imageCreateMaxArrayLayers` be defined analogously to `imageCreateMaxMipLevels`.

- Let `VkExtent3D imageCreateMaxExtent` be defined analogously to `imageCreateMaxMipLevels`.

- Let `VkSampleCountFlags imageCreateSampleCounts` be defined analogously to `imageCreateMaxMipLevels`. 
Valid Usage

- Each of the following values (as described in Image Creation Limits) must not be undefined: imageCreateMaxMipLevels, imageCreateMaxArrayLayers, imageCreateMaxExtent, and imageCreateSampleCounts.

- If sharingMode is VK_SHARING_MODE_CONCURRENT, pQueueFamilyIndices must be a valid pointer to an array of queueFamilyIndexCount uint32_t values.

- If sharingMode is VK_SHARING_MODE_CONCURRENT, queueFamilyIndexCount must be greater than 1.

- If sharingMode is VK_SHARING_MODE_CONCURRENT, each element of pQueueFamilyIndices must be unique and must be less than pQueueFamilyPropertyCount returned by vkGetPhysicalDeviceQueueFamilyProperties for the physicalDevice that was used to create device.

- format must not be VK_FORMAT_UNDEFINED.

- extent.width must be greater than 0.

- extent.height must be greater than 0.

- extent.depth must be greater than 0.

- mipLevels must be greater than 0.

- arrayLayers must be greater than 0.

- If flags contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, imageType must be VK_IMAGE_TYPE_2D.

- 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 `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 `VkPhysicalDeviceLimits::maxFramebufferWidth`

- 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`

- 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`

- `samples` must be a bit value that is set in `imageCreateSampleCounts` (as defined in Image Creation Limits)

- 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`

- If the sparse bindings feature is not enabled, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_BINDING_BIT`

- If the sparse aliased residency feature is not enabled, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_ALIASED_BIT`

- If `imageType` is `VK_IMAGE_TYPE_1D`, `flags` must not contain `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT`

- 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`

- 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`

- 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
- initialLayout must be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED

**Valid Usage (Implicit)**

- sType must be VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO
- pNext must be NULL
- flags must be a valid combination of VkImageCreateFlagBits values
- imageType must be a valid VkImageType value
- format must be a valid VkFormat value
- samples must be a valid VkSampleCountFlagBits value
- tiling must be a valid VkImageTiling value
- usage must be a valid combination of VkImageUsageFlagBits values
- usage must not be 0
- sharingMode must be a valid VkSharingMode value
- initialLayout must be a valid VkImageLayout value

Bits which can be set in VkImageCreateInfo::usage, specifying intended usage of an image, are:

```c
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,
} 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 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.

typedef VkFlags VkImageUsageFlags;

VkImageUsageFlags is a bitmask type for setting a mask of zero or more VkImageUsageFlagBits.

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,
} 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.

• 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.
See Sparse Resource Features and Sparse Physical Device Features for more details.

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:

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:

typedef enum VkImageTiling {
    VK_IMAGE_TILING_OPTIMAL = 0,
    VK_IMAGE_TILING_LINEAR = 1,
} VkImageTiling;

• 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).

To query the memory layout of an image subresource, call:

void vkGetImageSubresourceLayout(
    VkDevice device,              // device
    VkImage image,                // image
    const VkImageSubresource* pSubresource, // pSubresource
    VkSubresourceLayout* pLayout);
• \texttt{pLayout} is a pointer to a \texttt{VkSubresourceLayout} structure in which the layout is returned.

The image \textbf{must} be \textit{linear}. The returned layout is valid for \textit{host access}.

\texttt{vkGetImageSubresourceLayout} is invariant for the lifetime of a single image.

### Valid Usage

- \textbf{image must} have been created with \textit{tiling} equal to \texttt{VK_IMAGE_TILING_LINEAR}
- The \texttt{aspectMask} member of \texttt{pSubresource must} only have a single bit set
- The \texttt{mipLevel} member of \texttt{pSubresource must} be less than the \texttt{mipLevels} specified in \texttt{VkImageCreateInfo} when \texttt{image} was created
- The \texttt{arrayLayer} member of \texttt{pSubresource must} be less than the \texttt{arrayLayers} specified in \texttt{VkImageCreateInfo} when \texttt{image} was created

### Valid Usage (Implicit)

- \textbf{device must} be a valid \texttt{VkDevice} handle
- \textbf{image must} be a valid \texttt{VkImage} handle
- \textbf{pSubresource must} be a valid pointer to a valid \texttt{VkImageSubresource} structure
- \textbf{pLayout must} be a valid pointer to a \texttt{VkSubresourceLayout} structure
- \textbf{image must} have been created, allocated, or retrieved from \texttt{device}

The \texttt{VkImageSubresource} structure is defined as:

```c
typedef struct VkImageSubresource {
    VkImageAspectFlags aspectMask;
    uint32_t mipLevel;
    uint32_t arrayLayer;
} VkImageSubresource;
```

- \texttt{aspectMask} is a \texttt{VkImageAspectFlags} selecting the image \textit{aspect}.
- \texttt{mipLevel} selects the mipmap level.
- \texttt{arrayLayer} selects the array layer.

### Valid Usage (Implicit)

- \textbf{aspectMask must} be a valid combination of \texttt{VkImageAspectFlagBits} values
- \textbf{aspectMask must} not be \texttt{0}

Information about the layout of the image subresource is returned in a \texttt{VkSubresourceLayout}
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 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 color format, then the aspectMask member of VkImageSubresource must be VK_IMAGE_ASPECT_COLOR_BIT.

If the image has a depth/stencil format, 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.

To destroy an image, call:

```c
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

- All submitted commands that refer to `image`, either directly or via a `VkImageView`, must have completed execution
- If `VkAllocationCallbacks` were provided when `image` was created, a compatible set of callbacks must be provided here
- If no `VkAllocationCallbacks` were provided when `image` was created, `pAllocator` must be `NULL`

### Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle
- If `image` is not `VK_NULL_HANDLE`, `image` must be a valid `VkImage` handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- 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

### 11.3.1. 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.
11.3.2. Image Mipmap Sizing

A complete mipmap chain is the full set of mipmap levels, from the largest mipmap level provided, down to the minimum mipmap level.

Conventional Images

For conventional images, the dimensions of each successive mipmap level, $n+1$, are:

$$\text{width}_{n+1} = \max\left(\left\lfloor \frac{\text{width}_n}{2} \right\rfloor, 1\right)$$

$$\text{height}_{n+1} = \max\left(\left\lfloor \frac{\text{height}_n}{2} \right\rfloor, 1\right)$$

$$\text{depth}_{n+1} = \max\left(\left\lfloor \frac{\text{depth}_n}{2} \right\rfloor, 1\right)$$

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:

- 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:

$$\left\lceil \log_2(\max(\text{width}_0, \text{height}_0, \text{depth}_0)) \right\rceil + 1$$

where $\text{width}_0$, $\text{height}_0$, and $\text{depth}_0$ are the dimensions of the largest (most detailed) mipmap level, $\emptyset$.

11.4. 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 `vkCmdPipelineBarrier` or a `vkCmdWaitEvents` command buffer command (see Image Memory Barriers), or as part of a subpass dependency within a render pass (see `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
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,
} VkImageLayout;
```

The type(s) of device access supported by each layout are:

- `VK_IMAGE_LAYOUT_UNDEFINED` does not support device access. This layout must only be used as the
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 

**11.4.1. Image Layout Matching Rules**

At the time that any command buffer command accessing an image executes on any queue, the

---

**initialLayout** member of **VkImageCreateInfo** or **VkAttachmentDescription**, or as the **oldLayout** in an image transition. When transitioning out of this layout, the contents of the memory are not guaranteed to be preserved.

- **VK_IMAGE_LAYOUT_PREINITIALIZED** does not support device access. This layout must only be used as the **initialLayout** member of **VkImageCreateInfo** or **VkAttachmentDescription**, or as the **oldLayout** in an image transition. When transitioning out of this layout, the contents of the memory are preserved. 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_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.

- **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.

- **VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL** must only be used as a read-only image in a shader (which can be read as a sampled image, combined image/sampler and/or input attachment). This layout is valid only for image subresources of images created with the **VK_IMAGE_USAGE_SAMPLED_BIT** or **VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT** usage bit enabled.

- **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL** must only be used as a source image of a transfer command (see the definition of **VK_PIPELINE_STAGE_TRANSFER_BIT**). This layout is valid only for image subresources of images created with the **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** usage bit enabled.

- **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 **VK_IMAGE_USAGE_TRANSFER_DST_BIT** 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).
layouts of the image subresources that are accessed must all match exactly the layout specified via the API controlling those accesses.

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 be 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.

11.5. 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
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkImageView)
```

The types of image views that can be created are:

```c
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:

```c
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)

- **device** must be a valid `VkDevice` handle
- **pCreateInfo** must be a valid pointer to a valid `VkImageViewCreateInfo` structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid `VkAllocationCallbacks` structure
- **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:

```c
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 an extension-specific 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.

If **image** was created with the `VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT` flag, **format** can be different from the image's format, but if 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](#).

---

### Table 8. 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>
<tbody>
<tr>
<td></td>
<td><strong>imageType = ci.imageType</strong></td>
<td><strong>baseArrayLayer</strong> and <strong>layerCount</strong> are members of the <strong>subresourceRange</strong> member.</td>
</tr>
<tr>
<td></td>
<td><strong>width = ci.extent.width</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>height = ci.extent.height</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>depth = ci.extent.depth</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>arrayLayers = ci.arrayLayers</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>samples = ci.samples</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>flags = ci.flags</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>where ci is the <a href="#">VkImageCreateInfo</a> used to create <strong>image</strong>.</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>
| **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 |
| **2D, 0, 0**    | imageType = VK_IMAGE_TYPE_2D  
width ≥ 1  
height ≥ 1  
depth = 1  
arrayLayers ≥ 1  
samples = 1 | viewType = VK_IMAGE_VIEW_TYPE_2D  
baseArrayLayer ≥ 0  
layerCount = 1 |
| **2D, 1, 0**    | imageType = VK_IMAGE_TYPE_2D  
width ≥ 1  
height ≥ 1  
depth = 1  
arrayLayers ≥ 1  
samples = 1 | viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY  
baseArrayLayer ≥ 0  
layerCount ≥ 1 |
| **2D, 0, 1**    | imageType = VK_IMAGE_TYPE_2D  
width ≥ 1  
height ≥ 1  
depth = 1  
arrayLayers ≥ 1  
samples > 1 | viewType = VK_IMAGE_VIEW_TYPE_2D  
baseArrayLayer ≥ 0  
layerCount = 1 |
| **2D, 1, 1**    | imageType = VK_IMAGE_TYPE_2D  
width ≥ 1  
height ≥ 1  
depth = 1  
arrayLayers ≥ 1  
samples > 1 | viewType = VK_IMAGE_VIEW_TYPE_2D_ARRAY  
baseArrayLayer ≥ 0  
layerCount ≥ 1 |
<table>
<thead>
<tr>
<th>Dim, Arrayed, MS</th>
<th>Image parameters</th>
<th>View parameters</th>
</tr>
</thead>
</table>
| **CUBE, 0, 0**   | \( \text{imageType} = \text{VK\_IMAGE\_TYPE\_2D} \)  
width \( \geq 1 \)  
height = width  
depth = 1  
arrayLayers \( \geq 6 \)  
samples = 1  
flags includes \( \text{VK\_IMAGE\_CREATE\_CUBE\_COMPATIBLE\_BIT} \) | viewType = \( \text{VK\_IMAGE\_VIEW\_TYPE\_CUBE} \)  
baseArrayLayer \( \geq 0 \)  
layerCount = 6 |}

| **CUBE, 1, 0**   | \( \text{imageType} = \text{VK\_IMAGE\_TYPE\_2D} \)  
width \( \geq 1 \)  
height = width  
depth = 1  
\( N \geq 1 \)  
arrayLayers \( \geq 6 \times N \)  
samples = 1  
flags includes \( \text{VK\_IMAGE\_CREATE\_CUBE\_COMPATIBLE\_BIT} \) | viewType = \( \text{VK\_IMAGE\_VIEW\_TYPE\_CUBE\_ARRAY} \)  
baseArrayLayer \( \geq 0 \)  
layerCount = \( 6 \times N, N \geq 1 \) |}

| **3D, 0, 0**     | \( \text{imageType} = \text{VK\_IMAGE\_TYPE\_3D} \)  
width \( \geq 1 \)  
height \( \geq 1 \)  
depth \( \geq 1 \)  
arrayLayers = 1  
samples = 1 | viewType = \( \text{VK\_IMAGE\_VIEW\_TYPE\_3D} \)  
baseArrayLayer = 0  
layerCount = 1 |}
Valid Usage

• 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

• If the image cubemap arrays feature is not enabled, viewType must not be VK_IMAGE_VIEW_TYPE_CUBE_ARRAY

• image must have been created with a usage value containing at least one of VK_IMAGE_USAGE_SAMPLED_BIT, VK_IMAGE_USAGE_STORAGE_BIT, VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT

• The format features of the resultant image view must contain at least one bit

• 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

• If usage contains VK_IMAGE_USAGE_STORAGE_BIT, then the image view's format features must contain VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT

• If usage contains VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, then the image view's format features must contain VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT

• 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

• 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

• subresourceRange.baseMipLevel must be less than the mipLevels specified in VkImageCreateInfo when image was created

• 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

• subresourceRange.baseArrayLayer must be less than the arrayLayers specified in VkImageCreateInfo when image was created

• 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

• If image was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, format must be compatible with the format used to create image, as defined in Format Compatibility Classes

• If image was not created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, format must be identical to the format used to create image

• If image is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object

• subresourceRange and viewType must be compatible with the image, as described in the compatibility table
• If `viewType` is `VK_IMAGE_VIEW_TYPE_CUBE` and `subresourceRange.layerCount` is not `VK_REMAINING_ARRAY_LAYERS`, `subresourceRange.layerCount` must be 6

• 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

• If `viewType` is `VK_IMAGE_VIEW_TYPE_CUBE` and `subresourceRange.layerCount` is `VK_REMAINING_ARRAY_LAYERS`, the remaining number of layers must be 6

• 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

Valid Usage (Implicit)

• `sType` must be `VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO`

• `pNext` must be `NULL`

• `flags` must be 0

• `image` must be a valid `VkImage` handle

• `viewType` must be a valid `VkImageViewType` value

• `format` must be a valid `VkFormat` value

• `components` must be a valid `VkComponentMapping` structure

• `subresourceRange` must be a valid `VkImageSubresourceRange` structure

Bits which can be set in `VkImageViewCreateInfo::flags`, specifying additional parameters of an image, are:

```c
typedef enum VkImageViewCreateFlagBits {
} VkImageViewCreateFlagBits;
```

```c
typedef VkFlags VkImageViewCreateFlags;
```

`VkImageViewCreateFlags` is a bitmask type for setting a mask of zero or more `VkImageViewCreateFlagBits`.

The `VkImageSubresourceRange` structure is defined as:

```c
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. 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 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.

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 identity for storage image descriptors, input attachment descriptors, and framebuffer attachments.

---

**Valid Usage**

- If **levelCount** is not **VK_REMAINING_MIP_LEVELS**, it **must** be greater than 0
- If **layerCount** is not **VK_REMAINING_ARRAY_LAYERS**, it **must** be greater than 0

---

276 | Chapter 11. Resource Creation
Valid Usage (Implicit)

- `aspectMask` must be a valid combination of `VkImageAspectFlagBits` values
- `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:

```c
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,
} 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.

```c
typedef VkFlags VkImageAspectFlags;
```

`VkImageAspectFlags` is a bitmask type for setting a mask of zero or more `VkImageAspectFlagBits`.

The `VkComponentMapping` structure is defined as:

```c
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)

- \(r\) must be a valid \(VkComponentSwizzle\) value
- \(g\) must be a valid \(VkComponentSwizzle\) value
- \(b\) must be a valid \(VkComponentSwizzle\) value
- \(a\) 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:

```c
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>
</tbody>
</table>
To destroy an image view, call:

```c
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

- All submitted commands that refer to **imageView** must have completed execution.
- If **VkAllocationCallbacks** were provided when **imageView** was created, a compatible set of callbacks must be provided here.
- If no **VkAllocationCallbacks** were provided when **imageView** was created, **pAllocator** must be **NULL**.

### Valid Usage (Implicit)

- **device** must be a valid **VkDevice** handle.
- If **imageView** is not **VK_NULL_HANDLE**, **imageView** must be a valid **VkImageView** handle.
- If **pAllocator** is not **NULL**, **pAllocator** must be a valid pointer to a valid **VkAllocationCallbacks** structure.
- 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.

#### 11.5.1. 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, 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.

### 11.6. 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.

To determine the memory requirements for a buffer resource, call:

```c
void vkGetBufferMemoryRequirements(
    VkDevice device,        
    VkBuffer buffer,        
    VkMemoryRequirements* 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)

- `device` must be a valid `VkDevice` handle
- `buffer` must be a valid `VkBuffer` handle
- `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements` structure
- `buffer` must have been created, allocated, or retrieved from `device`

To determine the memory requirements for an image resource, call:

```c
void vkGetImageMemoryRequirements(
    VkDevice device,           // device is the logical device that owns the image.
    VkImage image,             // image is the image to query.
    VkMemoryRequirements*     // pMemoryRequirements is a pointer to a VkMemoryRequirements structure in which the memory requirements of the image object are returned.
    pMemoryRequirements);
```

Valid Usage

Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle
- `image` must be a valid `VkImage` handle
- `pMemoryRequirements` must be a valid pointer to a `VkMemoryRequirements` structure
- `image` must have been created, allocated, or retrieved from `device`

The `VkMemoryRequirements` structure is defined as:

```c
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 implementation guarantees certain properties about the memory requirements returned by `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.

• 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 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`, 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, 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, and the `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` of the `usage` member in the `VkImageCreateInfo` structure passed to `vkCreateImage`. 

282 | Chapter 11. Resource Creation
• 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.

  **Note**
  The implication of this requirement is that lazily allocated memory is disallowed for buffers in all cases.

• 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.

To attach memory to a buffer object, call:

```c
#include <vulkan/vulkan.h>

VkResult vkBindBufferMemory(
    VkDevice device,         // device
    VkBuffer buffer,        // buffer
    VkDeviceMemory memory,  // memory
    VkDeviceSize memoryOffset);
```

• **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.
Valid Usage

- `buffer` must not already be backed by a memory object
- `buffer` must not have been created with any sparse memory binding flags
- `memoryOffset` must be less than the size of `memory`
- `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`
- `memoryOffset` must be an integer multiple of the `alignment` member of the `VkMemoryRequirements` structure returned from a call to `vkGetBufferMemoryRequirements` with `buffer`
- 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`

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`

To attach memory to an image object, call:
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.

### Valid Usage

• **image** must not already be backed by a memory object
• **image** must not have been created with any sparse memory binding flags
• **memoryOffset** must be less than the size of **memory**
• **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**
• **memoryOffset** must be an integer multiple of the **alignment** member of the **VkMemoryRequirements** structure returned from a call to **vkGetImageMemoryRequirements** with **image**
• 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)

• **device** must be a valid **VkDevice** handle
• **image** must be a valid **VkImage** handle
• **memory** must be a valid **VkDeviceMemory** handle
• **image** must have been created, allocated, or retrieved from **device**
• **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

**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:

\[
\begin{align*}
resourceA.end & \quad = \quad resourceA.memoryOffset + resourceA.size - 1 \\
resourceA.endPage & \quad = \quad resourceA.end \& \neg(bufferImageGranularity-1) \\
resourceB.start & \quad = \quad resourceB.memoryOffset \\
resourceB.startPage & \quad = \quad resourceB.start \& \neg(bufferImageGranularity-1)
\end{align*}
\]

The following property **must** hold:

\[
resourceA.endPage < resourceB.startPage
\]

That is, the end of the first resource (A) and the beginning of the second resource (B) **must** be on separate “pages” of size `bufferImageGranularity`. `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 `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 `bufferImageGranularity`. Therefore, memory bound to sparse resources naturally satisfies the `bufferImageGranularity`.  

286 | Chapter 11. Resource Creation
11.7. Resource Sharing Mode

Buffer and image objects are created with a *sharing mode* controlling how they can be accessed from queues. The supported sharing modes are:

```c
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

- **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.
- **VK_SHARING_MODE_CONCURRENT** specifies that concurrent access to any range or image subresource of the object from multiple queue families is supported.

*Note*  
**VK_SHARING_MODE_CONCURRENT** may result in lower performance access to the buffer or image than **VK_SHARING_MODE_EXCLUSIVE**.

Ranges of buffers and image subresources of image objects created using **VK_SHARING_MODE_EXCLUSIVE** must only be accessed by queues in the queue family that has *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 **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.

*Note*  
Images still require a *layout transition* from **VK_IMAGE_LAYOUT_UNDEFINED** or **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.

11.8. Memory Aliasing

A range of a *VkDeviceMemory* allocation is aliased if it is bound to multiple resources simultaneously, as described below, via *vkBindImageMemory*, *vkBindBufferMemory*, or via *sparse memory bindings*.
Consider two resources, resource_A and resource_B, bound respectively to memory range_A and range_B. Let paddedRange_A and paddedRange_B be, respectively, range_A and range_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_A and range_B. If one resource is linear and the other is non-linear, then the resources alias the memory in the intersection of paddedRange_A and paddedRange_B.

Applications can alias memory, but use of multiple aliases is subject to several constraints.

Note

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.

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 not 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.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>
Chapter 12. 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:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkSampler)
```

To create a sampler object, call:

```
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 (Implicit)

- `device` must be a valid `VkDevice` handle
- `pCreateInfo` must be a valid pointer to a valid `VkSamplerCreateInfo` structure
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- `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`
- `VK_ERROR_TOO_MANY_OBJECTS`
The **VkSamplerCreateInfo** structure is defined as:

```c
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 an extension-specific 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` and `maxLod` are the values used to clamp the computed LOD value, as described in the Level-of-Detail Operation section.

• `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. If `maxSamplerAllocationCount` is exceeded, `vkCreateSampler` will return `VK_ERROR_TOO_MANY_OBJECTS`. 

---

292 | Chapter 12. Samplers
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

- The absolute value of `mipLodBias` must be less than or equal to `VkPhysicalDeviceLimits::maxSamplerLodBias`
- `maxLod` must be greater than or equal to `minLod`
- If the anisotropic sampling feature is not enabled, `anisotropyEnable` must be `VK_FALSE`
- If `anisotropyEnable` is `VK_TRUE`, `maxAnisotropy` must be between 1.0 and `VkPhysicalDeviceLimits::maxSamplerAnisotropy`, inclusive
- If `unnormalizedCoordinates` is `VK_TRUE`, `minFilter` and `magFilter` must be equal
- If `unnormalizedCoordinates` is `VK_TRUE`, `mipmapMode` must be `VK_SAMPLER_MIPMAP_MODE_NEAREST`
- If `unnormalizedCoordinates` is `VK_TRUE`, `minLod` and `maxLod` must be zero
- 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`
- If `unnormalizedCoordinates` is `VK_TRUE`, `anisotropyEnable` must be `VK_FALSE`
- If `unnormalizedCoordinates` is `VK_TRUE`, `compareEnable` must be `VK_FALSE`
- If any of `addressModeU`, `addressModeV` or `addressModeW` are `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`, `borderColor` must be a valid `VkBorderColor` value
- If `compareEnable` is `VK_TRUE`, `compareOp` must be a valid `VkCompareOp` value

### Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- `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 {
} VkSamplerCreateFlagBits;

typedef VkFlags VkSamplerCreateFlags;

VkSamplerCreateFlags is a bitmask type for setting a mask of zero or more VkSamplerCreateFlagBits.

Possible values of the VkSamplerCreateInfo::magFilter and minFilter parameters, specifying filters used for texture lookups, are:

typedef enum VkFilter {
    VK_FILTER_NEAREST = 0,
    VK_FILTER_LINEAR = 1,
} VkFilter;

- VK_FILTER_NEAREST specifies nearest filtering.
- VK_FILTER_LINEAR specifies linear 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,
} 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.

Possible values of *VkSamplerCreateInfo::borderColor*, specifying the border color used for texture lookups, are:

```c
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,
} 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.

These colors are described in detail in *Texel Replacement*.

To destroy a sampler, call:

```c
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

• All submitted commands that refer to `sampler` must have completed execution

• If `VkAllocationCallbacks` were provided when `sampler` was created, a compatible set of callbacks must be provided here

• If no `VkAllocationCallbacks` were provided when `sampler` was created, `pAllocator` must be `NULL`

Valid Usage (Implicit)

• `device` must be a valid `VkDevice` handle

• If `sampler` is not `VK_NULL_HANDLE`, `sampler` must be a valid `VkSampler` handle

• If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure

• 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
Chapter 13. 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.

13.1. 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.

13.1.1. 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_GENERAL layout in order to access its data in a shader.

13.1.2. 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.

13.1.3. 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_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.

13.1.4. 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.

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_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.

13.1.5. 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.
### 13.1.6. 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 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 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.

### 13.1.7. 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.

### 13.1.8. 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.

### 13.1.9. 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
13.1.10. 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.

13.1.11. 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 the 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.

13.2. 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.

13.2.1. 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
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSetLayout)
```

To create descriptor set layout objects, call:
VkResult vkCreateDescriptorSetLayout(
    VkDevice                                    device,
    const VkDescriptorSetLayoutCreateInfo*      pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorSetLayout*                      pSetLayout);

- **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)

- **device** **must** be a valid `VkDevice` handle
- **pCreateInfo** **must** be a valid pointer to a valid `VkDescriptorSetLayoutCreateInfo` structure
- If **pAllocator** is not `NULL`, **pAllocator** **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- **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
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 an extension-specific structure.
• `flags` is a bitmask 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**

- The `VkDescriptorSetLayoutBinding::binding` members of the elements of the `pBindings` array must each have different values.

**Valid Usage (Implicit)**

- `sType` must be `VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- If `bindingCount` is not `0`, `pBindings` must be a valid pointer to an array of `bindingCount` valid `VkDescriptorSetLayoutBinding` structures

Bits which can be set in `VkDescriptorSetLayoutCreateInfo::flags` to specify options for descriptor set layout are:

```c
typedef enum VkDescriptorSetLayoutCreateFlagBits {
} VkDescriptorSetLayoutCreateFlagBits;
```

**Note**

All bits for this type are defined by extensions, and none of those extensions are enabled in this build of the specification.

```c
typedef VkFlags VkDescriptorSetLayoutCreateFlags;
```

`VkDescriptorSetLayoutCreateFlags` is a bitmask type for setting a mask of zero or more `VkDescriptorSetLayoutCreateFlagBits`.

The `VkDescriptorSetLayoutBinding` structure is defined as:
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. 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 resource.

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 are copied; the sampler objects **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

- 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.

- If `descriptorCount` is not 0, `stageFlags` must be a valid combination of `VkShaderStageFlagBits` values.

- If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT` and `descriptorCount` is not 0, then `stageFlags` must be 0 or `VK_SHADER_STAGE_FRAGMENT_BIT`.

---

### Valid Usage (Implicit)

- `descriptorType` must be a valid `VkDescriptorType` value.

The following examples show a shader snippet using two descriptor sets, and application code that creates corresponding descriptor set layouts.

**GLSL example**

```glsl

// 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];
};
```

---

304 | Chapter 13. Resource Descriptors
SPIR-V example

```spir-v
...%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

```c
VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] = {
    // binding to a single image descriptor
    { 0,
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, // descriptorType
        1, // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT, // stageFlags
    },
    ...
```

Chapter 13. Resource Descriptors | 305
const VkDescriptorSetLayoutCreateInfo myDescriptorSetLayoutCreateInfo[] = {
    {VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO, NULL, 0, 2, &myDescriptorSetLayoutBinding[0]}
};

const VkDescriptorSetLayout myDescriptorSetLayout[2];

// Create first descriptor set layout
// myResult = vkCreateDescriptorSetLayout(
//     myDevice,
//     myDescriptorSetLayoutCreateInfo[0],
To destroy a descriptor set layout, call:

```c
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

- If `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, a compatible set of callbacks must be provided here.
- If no `VkAllocationCallbacks` were provided when `descriptorSetLayout` was created, `pAllocator` must be NULL.

### Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle.
- If `descriptorSetLayout` is not `VK_NULL_HANDLE`, `descriptorSetLayout` must be a valid `VkDescriptorSetLayout` handle.
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure.
- 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

13.2.2. 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:

```cpp
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkPipelineLayout)
```

To create a pipeline layout, call:

```cpp
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)

- **device** must be a valid VkDevice handle
- **pCreateInfo** must be a valid pointer to a valid VkPipelineLayoutCreateInfo structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid VkAllocationCallbacks structure
- **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
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 an extension-specific 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

- `setLayoutCount` must be less than or equal to `VkPhysicalDeviceLimits::maxBoundDescriptorSets`

- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_SAMPLER` and `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER` accessible to any shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSamplers`

- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER` and `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` accessible to any shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorUniformBuffers`

- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` accessible to any shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageBuffers`

- 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 to any shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorSampledImages`

- The total number of descriptors of the type `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`, and `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER` accessible to any shader stage across all elements of `pSetLayouts` must be less than or equal to `VkPhysicalDeviceLimits::maxPerStageDescriptorStorageImages`

- The total number of descriptors of the type `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`

- 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 `VkPhysicalDeviceLimits::maxDescriptorSetSamplers`

- 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 `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers`

- 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 `VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic`

- 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 `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffers`

- 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 `VkPhysicalDeviceLimits::maxDescriptorSetStorageBuffersDynamic`
• 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 VkPhysicalDeviceLimits::maxDescriptorSetSampledImages.

• 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 VkPhysicalDeviceLimits::maxDescriptorSetStorageImages.

• 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 VkPhysicalDeviceLimits::maxDescriptorSetInputAttachments.

• Any two elements of pPushConstantRanges must not include the same stage in stageFlags.

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.

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:

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

- **offset must** be less than `VkPhysicalDeviceLimits::maxPushConstantsSize`
- **offset must** be a multiple of 4
- **size must** be greater than 0
- **size must** be a multiple of 4
- **size must** be less than or equal to `VkPhysicalDeviceLimits::maxPushConstantsSize` minus `offset`

### Valid Usage (Implicit)

- **stageFlags must** be a valid combination of `VkShaderStageFlagBits` values
- **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 10. Pipeline Layout Resource Limits

<table>
<thead>
<tr>
<th>Total Resources Available</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxDescriptorSetSamplers</td>
<td>sampler</td>
</tr>
<tr>
<td></td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxDescriptorSetSampledImages</td>
<td>sampled image</td>
</tr>
<tr>
<td></td>
<td>combined image sampler</td>
</tr>
<tr>
<td></td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>storage image</td>
</tr>
<tr>
<td></td>
<td>storage texel buffer</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffers</td>
<td>uniform buffer</td>
</tr>
<tr>
<td></td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetUniformBuffersDynamic</td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffers</td>
<td>storage buffer</td>
</tr>
<tr>
<td>maxDescriptorSetStorageBuffersDynamic</td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments</td>
<td>input attachment</td>
</tr>
</tbody>
</table>

To destroy a pipeline layout, call:

```c
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**

- If `VkAllocationCallbacks` were provided when `pipelineLayout` was created, a compatible set of callbacks **must** be provided here.
- If no `VkAllocationCallbacks` were provided when `pipelineLayout` was created, `pAllocator` **must** be `NULL`.
- `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)

- **device must** be a valid `VkDevice` handle
- If `pipelineLayout` is not `VK_NULL_HANDLE`, `pipelineLayout` **must** be a valid `VkPipelineLayout` handle
- If `pAllocator` is not NULL, `pAllocator` **must** be a valid pointer to a valid `VkAllocationCallbacks` structure
- 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 previous 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

```cpp
const VkDescriptorSetLayout layouts[] = { layout1, layout2 }; const VkPushConstantRange ranges[] = {
    {
        VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,  // stageFlags
        0,                                      // offset
        4                                       // size
    },
    {
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_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);
```
13.2.3. 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:

```cpp
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorPool)
```

To create a descriptor pool object, call:

```cpp
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.

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

The created descriptor pool is returned in **pDescriptorPool**.

### Valid Usage (Implicit)

- **device** must be a valid VkDevice handle
- **pCreateInfo** must be a valid pointer to a valid VkDescriptorPoolCreateInfo structure
- If **pAllocator** is not NULL, **pAllocator** must be a valid pointer to a valid VkAllocationCallbacks structure
- **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

Additional information about the pool is passed in a `VkDescriptorPoolCreateInfo` structure:

```c
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 an extension-specific 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.

Valid Usage

- **maxSets** must be greater than 0

Valid Usage (Implicit)

- **sType** must be VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO
- **pNext** must be NULL
- **flags** must be a valid combination of VkDescriptorPoolCreateFlagBits values
- **pPoolSizes** must be a valid pointer to an array of poolSizeCount valid VkDescriptorPoolSize structures
- **poolSizeCount** must be greater than 0

Bits which can be set in VkDescriptorPoolCreateInfo::flags to enable operations on a descriptor pool are:

```c
typedef enum VkDescriptorPoolCreateFlagBits {
  VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT = 0x00000001,
} 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.

```c
typedef VkFlags VkDescriptorPoolCreateFlags;
```

VkDescriptorPoolCreateFlags is a bitmask type for setting a mask of zero or more VkDescriptorPoolCreateFlagBits.

The VkDescriptorPoolSize structure is defined as:

```c
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.

### Valid Usage

- **descriptorCount** must be greater than 0

### Valid Usage (Implicit)

- **type** must be a valid `VkDescriptorType` value

To destroy a descriptor pool, call:

```c
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

- All submitted commands that refer to **descriptorPool** (via any allocated descriptor sets) must have completed execution
- If `VkAllocationCallbacks` were provided when **descriptorPool** was created, a compatible set of callbacks must be provided here
- If no `VkAllocationCallbacks` were provided when **descriptorPool** was created, **pAllocator** must be NULL

---

Chapter 13. Resource Descriptors | 319
Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle
- If `descriptorPool` is not `VK_NULL_HANDLE`, `descriptorPool` must be a valid `VkDescriptorPool` handle
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- 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:

\[
\text{VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkDescriptorSet)}
\]

To allocate descriptor sets from a descriptor pool, call:

\[
\text{VkResult vkAllocateDescriptorSets(}
\text{    VkDevice device,}
\text{    const VkDescriptorSetAllocateInfo* pAllocateInfo,}
\text{    VkDescriptorSet* pDescriptorSets);}
\]

- **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:

- Descriptors that are statically used must have been populated before the descriptor set is consumed.
- Entries that are not used by a pipeline can have undefined descriptors.
If an allocation fails due to fragmentation, an indeterminate error is returned with an unspecified error code. Any returned error other than `VK_ERROR_FRAGMENTED_POOL` does not imply its usual meaning: applications **should** assume that the allocation failed due to fragmentation, and create a new descriptor pool.

*Note*
Applications **should** check for a negative return value when allocating new descriptor sets, assume that any error effectively means `VK_ERROR_FRAGMENTED_POOL`, and try to create a new descriptor pool. If `VK_ERROR_FRAGMENTED_POOL` is the actual return value, it adds certainty to that decision.

The reason for this is that `VK_ERROR_FRAGMENTED_POOL` was only added in a later version of the 1.0 specification, and so drivers **may** return other errors if they were written against earlier versions. To ensure full compatibility with earlier patch versions, these other errors are allowed.

---

**Valid Usage (Implicit)**

- **device** must be a valid `VkDevice` handle
- **pAllocateInfo** must be a valid pointer to a valid `VkDescriptorSetAllocateInfo` structure
- **pDescriptorSets** must be a valid pointer to an array of `pAllocateInfo->descriptorSetCount` `VkDescriptorSet` handles
- The value referenced by `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`

The `VkDescriptorSetAllocateInfo` structure is defined as:
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 an extension-specific 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

- **descriptorSetCount** must not be greater than the number of sets that are currently available for allocation in **descriptorPool**
- **descriptorPool** must have enough free descriptor capacity remaining to allocate the descriptor sets of the specified layouts

### Valid Usage (Implicit)

- **sType** must be **VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO**
- **pNext** must be NULL
- **descriptorPool** must be a valid **VkDescriptorPool** handle
- **pSetLayouts** must be a valid pointer to an array of **descriptorSetCount** valid **VkDescriptorSetLayout** handles
- **descriptorSetCount** must be greater than 0
- Both of **descriptorPool**, and the elements of **pSetLayouts** must have been created, allocated, or retrieved from the same **VkDevice**

To free allocated descriptor sets, call:

```c
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

- All submitted commands that refer to any element of **pDescriptorSets** must have completed execution.

- **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**.

- Each valid handle in **pDescriptorSets** must have been allocated from **descriptorPool**.

- **descriptorPool** must have been created with the **VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT** flag.

---

### Valid Usage (Implicit)

- **device** must be a valid **VkDevice** handle.

- **descriptorPool** must be a valid **VkDescriptorPool** handle.

- **descriptorSetCount** must be greater than 0.

- **descriptorPool** must have been created, allocated, or retrieved from **device**.

- 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:
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**

- All uses of `descriptorPool` (via any allocated descriptor sets) **must** have completed execution

**Valid Usage (Implicit)**

- `device` **must** be a valid `VkDevice` handle
- `descriptorPool` **must** be a valid `VkDescriptorPool` handle
- `flags` **must** be `0`
- `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`

```c
typedef VkFlags VkDescriptorPoolResetFlags;
```

`VkDescriptorPoolResetFlags` is a bitmask type for setting a mask, but is currently reserved for future use.
13.2.4. Descriptor Set Updates

Once allocated, descriptor sets can be updated with a combination of write and copy operations. To update descriptor sets, call:

```c
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**, that command buffer becomes **invalid**.

### Valid Usage

- The **dstSet** member of each element of **pDescriptorWrites** or **pDescriptorCopies** must not be used by any command that was recorded to a command buffer which is in the **pending** state.
 Valid Usage (Implicit)

- **device** must be a valid `VkDevice` handle

- If `descriptorWriteCount` is not 0, `pDescriptorWrites` must be a valid pointer to an array of `descriptorWriteCount` valid `VkWriteDescriptorSet` structures

- 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
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 an extension-specific 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.

- **descriptorCount** is the number of descriptors to update (the number of elements in `pImageInfo`, `pBufferInfo`, or `pTexelBufferView`).

- **descriptorType** is a `VkDescriptorType` specifying the type of each descriptor in `pImageInfo`, `pBufferInfo`, or `pTexelBufferView`, as described below. It **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 pBufferInfo, pBufferInfo, or pTexelBufferView members is used according to the descriptor type specified in the descriptorType member of the containing VkWriteDescriptorSet structure, as specified below.

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.
Valid Usage

• **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.

• **dstBinding must** be a binding with a non-zero **descriptorCount**.

• All consecutive bindings updated via a single `VkWriteDescriptorSet` structure, except those with a **descriptorCount** of zero, **must** have identical **descriptorType** and **stageFlags**.

• 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.

• **descriptorType must** match the type of **dstBinding** within **dstSet**.

• **dstSet must** be a valid `VkDescriptorSet` handle.

• 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**.

• 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.

• 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**.

• If **descriptorType** is **VK_DESCRIPTOR_TYPE_UNIFORM_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**, the **imageView** member of each element of **pImageInfo must** be either a valid `VkImageView` handle or **VK_NULL_HANDLE**.

• If **descriptorType** is **VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER**, **VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE**, **VK_DESCRIPTOR_TYPE_STORAGE_IMAGE**, and the **nullDescriptor** feature is not enabled, 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_STORAGE_IMAGE, for each descriptor that will be accessed via load or store operations the imageLayout member for corresponding elements of pImageInfo must be VK_IMAGE_LAYOUT_GENERAL

- If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the offset member of each element of pBufferInfo must be a multiple of VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment

- If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the offset member of each element of pBufferInfo must be a multiple of VkPhysicalDeviceLimits::minStorageBufferOffsetAlignment

- 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

- If descriptorType is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the buffer member of each element of pBufferInfo must have been created with VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT set

- If descriptorType is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the buffer member of each element of pBufferInfo must have been created with VK_BUFFER_USAGE_STORAGE_BUFFER_BIT set

- 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

- 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

- 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

- 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

- 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
• If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE` or `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `imageLayout` member of each element of `pImageInfo` must be a member of the list given in `Sampled Image` or `Combined Image Sampler`, corresponding to its type.

• If `descriptorType` is `VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT`, the `imageView` member of each element of `pImageInfo` must have been created with `VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT` set.

• 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.

• If `descriptorType` is `VK_DESCRIPTOR_TYPE_SAMPLER`, then `dstSet` must not have been allocated with a layout that included immutable samplers for `dstBinding`.

Valid Usage (Implicit)

• `sType` must be `VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET`.

• `pNext` must be `NULL`.

• `descriptorType` must be a valid `VkDescriptorType` value.

• `descriptorCount` must be greater than 0.

• 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
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,
} 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.

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.

The *VkDescriptorBufferInfo* structure is defined as:

```c
typedef struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
} VkDescriptorBufferInfo;
```

• *buffer* is 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

- offset must be less than the size of buffer
- If range is not equal to VK_WHOLE_SIZE, range must be greater than 0
- If range is not equal to VK_WHOLE_SIZE, range must be less than or equal to the size of buffer minus offset
- If the nullDescriptor feature is not enabled, buffer must not be VK_NULL_HANDLE

Valid Usage (Implicit)

- If buffer is not VK_NULL_HANDLE, buffer must be a valid VkBuffer handle

The VkDescriptorImageInfo structure is defined as:

```c
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 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

- 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.
- `imageLayout` 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.

Valid Usage (Implicit)

- Both of `imageView` and `sampler` that are valid handles of non-ignored parameters must have been created, allocated, or retrieved from the same `VkDevice`.

The `VkCopyDescriptorSet` structure is defined as:

```c
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 an extension-specific structure.
- `srcSet`, `srcBinding`, and `srcArrayElement` are the source set, binding, and array element, respectively.
- `dstSet`, `dstBinding`, and `dstArrayElement` are the destination set, binding, and array element, respectively.
- `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.
Valid Usage

- `srcBinding` must be a valid binding within `srcSet`
- 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
- `dstBinding` must be a valid binding within `dstSet`
- 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
- The type of `dstBinding` within `dstSet` must be equal to the type of `srcBinding` within `srcSet`
- 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
- 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`

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET`
- `pNext` must be `NULL`
- `srcSet` must be a valid `VkDescriptorSet` handle
- `dstSet` must be a valid `VkDescriptorSet` handle
- Both of `dstSet`, and `srcSet` must have been created, allocated, or retrieved from the same `VkDevice`

13.2.5. Descriptor Set Binding

To bind one or more descriptor sets to a command buffer, call:

```c
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 whether the descriptors will be used by graphics pipelines or compute pipelines. There is a separate set of bind points for each of graphics and compute, so binding one does not disturb the other.
• `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 write 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 `[firstSet.. firstSet+descriptorSetCount-1]` to use the bindings stored in `pDescriptorSets[0..descriptorSetCount-1]` for subsequent rendering commands (either compute or graphics, according to the `pipelineBindPoint`). Any bindings that were previously applied via these sets are no longer valid.

Once bound, a descriptor set affects rendering of subsequent graphics or compute commands in the command buffer until a different set is bound to the same set number, or else until 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 graphics or compute commands, 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:

• 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**

- 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`

- `dynamicOffsetCount` **must** be equal to the total number of dynamic descriptors in `pDescriptorSets`

- The sum of `firstSet` and `descriptorSetCount` **must** be less than or equal to `VkPipelineLayoutCreateInfo::setLayoutCount` provided when `layout` was created

- `pipelineBindPoint` **must** be supported by the commandBuffer's parent `VkCommandPool`'s queue family

- 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`

- 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`

- 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
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pipelineBindPoint` must be a valid `VkPipelineBindPoint` value
- `layout` must be a valid `VkPipelineLayout` handle
- `pDescriptorSets` must be a valid pointer to an array of `descriptorSetCount` valid `VkDescriptorSet` handles
- If `dynamicOffsetCount` is not 0, `pDynamicOffsets` must be a valid pointer to an array of `dynamicOffsetCount` `uint32_t` values
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations
- `descriptorSetCount` must be greater than 0
- 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>

13.2.6. 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.

The values of push constants are undefined at the start of a command buffer.
To update push constants, call:

```c
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.
```

### 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

- 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.
- 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`.
- `offset` **must** be a multiple of 4
- `size` **must** be a multiple of 4
- `offset` **must** be less than `VkPhysicalDeviceLimits::maxPushConstantsSize`
- `size` **must** be less than or equal to `VkPhysicalDeviceLimits::maxPushConstantsSize minus offset`
Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `layout` **must** be a valid `VkPipelineLayout` handle
- `stageFlags` **must** be a valid combination of `VkShaderStageFlagBits` values
- `stageFlags` **must** not be 0
- `pValues` **must** be a valid pointer to an array of `size` bytes
- `commandBuffer` **must** be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics, or compute operations
- `size` **must** be greater than 0
- 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>
Chapter 14. 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
- 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

This specification describes valid uses for Vulkan of these decorations. Any other use of one of these decorations is invalid.

14.1. 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.
14.1.1. 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.

14.1.2. 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. These variables must be identified with a Location decoration and can also be identified with a Component decoration.

14.1.3. Interface Matching

Interface matching rules only apply to built-ins when they are declared as members of the built-in interface block.

Tessellation control 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. For the purposes of interface matching, variables declared without a Component decoration are considered to have a Component decoration of zero.

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. 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 stages 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 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.

### 14.1.4. 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 32 or 64 are supported in shader interfaces.

Inputs and outputs of the following types consume a single interface location:

- 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 \( n \times m \) 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 `OpVariable` with a structure type that is not a block must be decorated with a `Location`.

When an `OpVariable` with a structure type (either block or non-block) is decorated with a `Location`, the members in the structure type must not be decorated with a `Location`. The `OpVariable`'s members are assigned consecutive locations in declaration order, starting from the first member, which is assigned the location decoration from the `OpVariable`.

When a block-type `OpVariable` is declared without a `Location` decoration, each member in its structure type must be decorated with a `Location`. Types nested deeper than the top-level members...
must not have 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 OpEntryPoint must not be assigned the same location, either explicitly or implicitly. Any two outputs listed as operands on the same OpEntryPoint 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 built-in interface block and the user-defined variable interface count against these limits. Each effective Location 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 11. 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>

14.1.5. 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 single precision 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.

### 14.2. 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.

### 14.3. 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 pColorAttachments\([i]\), 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. 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.

14.4. Fragment Input Attachment Interface

When a fragment stage is present in a pipeline, the fragment shader subpass inputs form an interface with the input attachments of the current subpass. The fragment shader subpass 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 subpass 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.
14.5. 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 resources that form the shader resource interface for a stage are the variables statically used by OpEntryPoint with the storage class of Uniform, UniformConstant, 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.

14.5.1. 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.

14.5.2. Descriptor Set Interface

The descriptor set interface is comprised of the shader variables with the storage class of 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.

The `Offset` decoration for any variable in a `Block` must not cause the space required for that variable to extend outside the range `[0, maxUniformBufferRange)`. The `Offset` decoration for any variable in a `BufferBlock` must not cause the space required for that variable to extend outside the range `[0, maxStorageBufferRange)`. 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`
- Sampled images (except uniform texel buffers), samplers and combined image samplers:
  - Dynamically uniform: `shaderSampledImageArrayDynamicIndexing` and
    `SampledImageArrayDynamicIndexing`
- Uniform buffers:
  - Dynamically uniform: `shaderUniformBufferArrayDynamicIndexing` and
    `UniformBufferArrayDynamicIndexing`
- Storage buffers:
  - Dynamically uniform: `shaderStorageBufferArrayDynamicIndexing` and
    `StorageBufferArrayDynamicIndexing`

If an instruction loads from or stores to a resource (including atomics and image instructions) and the resource descriptor being accessed is not uniform, then the corresponding dynamic indexing feature must be enabled and the capability must be declared.
Table 12. 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>
</tbody>
</table>

Table 13. 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>
<tr>
<td>storage image</td>
<td>UniformConstant</td>
<td>OpTypeImage (Sampled=2)</td>
<td></td>
</tr>
<tr>
<td>combined image sampler</td>
<td>UniformConstant</td>
<td>OpTypeSampledImage</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpTypeImage (Sampled=1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OpTypeSampler</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, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td>storage buffer</td>
<td>Uniform</td>
<td>OpTypeStruct</td>
<td>BufferBlock, Offset, (ArrayStride), (MatrixStride)</td>
</tr>
<tr>
<td>input attachment</td>
<td>UniformConstant</td>
<td>OpTypeImage (Dim =SubpassData, Sampled=2)</td>
<td>InputAttachmentIndex</td>
</tr>
</tbody>
</table>

¹ in addition to DescriptorSet and Binding

14.5.3. 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 \texttt{VkDescriptorSetLayoutBinding} that has a \texttt{binding} equal to \texttt{b} in \texttt{pSetLayouts[s]} that was specified in \texttt{VkPipelineLayoutCreateInfo}.

\texttt{DescriptorSet} decoration values \textbf{must} be between zero and \texttt{maxBoundDescriptorSets} minus one, inclusive. \texttt{Binding} decoration values \textbf{can} be any 32-bit unsigned integer value, as described in \texttt{Descriptor Set Layout}. Each descriptor set has its own binding name space.

If the \texttt{Binding} decoration is used with an array, the entire array is assigned that binding value. The array \textbf{must} be a single-dimensional array and size of the array \textbf{must} be no larger than the number of descriptors in the binding. The array \textbf{must} not be runtime-sized. The index of each element of the array is referred to as the \texttt{arrayElement}. For the purposes of interface matching and descriptor set \texttt{operations}, if a resource variable is not an array, it is treated as if it has an \texttt{arrayElement} of zero.

There is a limit on the number of resources of each type that \textbf{can} be accessed by a pipeline stage as shown in \texttt{Shader Resource Limits}. The “Resources Per Stage” column gives the limit on the number each type of resource that \textbf{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 pipeline layout \textbf{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 \texttt{stageFlags}.

However, if a variable assigned to a given \texttt{DescriptorSet} and \texttt{Binding} is statically used by the entry point for a shader stage, the pipeline layout \textbf{must} contain a descriptor set layout binding in that descriptor set layout and for that binding number, and that binding’s \texttt{stageFlags} \textbf{must} include the appropriate \texttt{VkShaderStageFlagBits} for that stage. The variable \textbf{must} be of a valid resource type determined by its SPIR-V type and storage class, as defined in \texttt{Shader Resource and Storage Class Correspondence}. The descriptor set layout binding \textbf{must} be of a corresponding descriptor type, as defined in \texttt{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`, some could be `OpTypeSampler` (`Sampled=1`), and some could be `OpTypeSampledImage`.

<table>
<thead>
<tr>
<th>Resources per Stage</th>
<th>Resource Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxPerStageDescriptorSamplers</td>
<td>sampler</td>
</tr>
<tr>
<td></td>
<td>combined image sampler</td>
</tr>
<tr>
<td>maxPerStageDescriptorSampledImages</td>
<td>sampled image</td>
</tr>
<tr>
<td></td>
<td>combined image sampler</td>
</tr>
<tr>
<td></td>
<td>uniform texel buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageImages</td>
<td>storage image</td>
</tr>
<tr>
<td></td>
<td>storage texel buffer</td>
</tr>
<tr>
<td>maxPerStageDescriptorUniformBuffers</td>
<td>uniform buffer</td>
</tr>
<tr>
<td></td>
<td>uniform buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorStorageBuffers</td>
<td>storage buffer</td>
</tr>
<tr>
<td></td>
<td>storage buffer dynamic</td>
</tr>
<tr>
<td>maxPerStageDescriptorInputAttachments</td>
<td>input attachment¹</td>
</tr>
</tbody>
</table>

¹ Input attachments can only be used in the fragment shader stage
14.5.4. Offset and Stride Assignment

All variables with a storage class of Uniform, StorageBuffer, or PushConstant must be explicitly laid out using the Offset, ArrayStride, and MatrixStride decorations.

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.

Standard Buffer Layout

Every member of an OpTypeStruct with storage class of Uniform, StorageBuffer, or PushConstant must be aligned according to the first matching rule as follows:

1. Any member of an OpTypeStruct with a storage class of Uniform and a decoration of Block must be aligned according to its extended alignment.
2. Every other member **must** be aligned according to its 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.
- 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.

### 14.6. 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.

**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 point is outside the clip half-space.

The **ClipDistance** decoration **must** be used only within vertex, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In vertex shaders, any variable decorated with **ClipDistance** **must** be declared using the **Output** storage class.

In fragment shaders, any variable decorated with **ClipDistance** **must** be declared using the **Input** storage class.

In tessellation control, tessellation evaluation, or geometry shaders, any variable decorated with **ClipDistance** **must not** be in a storage class other than **Input** or **Output**.

Any variable decorated with **ClipDistance** **must** be declared as an array of 32-bit floating-point values.
The array variable decorated with `ClipDistance` is explicitly sized by the shader.

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.

**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.

The `CullDistance` decoration must be used only within vertex, fragment, tessellation control, tessellation evaluation, and geometry shaders.

In vertex shaders, any variable decorated with `CullDistance` must be declared using the `Output` storage class.

In fragment shaders, any variable decorated with `CullDistance` must be declared using the `Input` storage class.

In tessellation control, tessellation evaluation, or geometry shaders, any variable decorated with `CullDistance` must not be declared in a storage class other than input or output.

Any variable decorated with `CullDistance` must be declared as an array of 32-bit floating-point values.

In fragment shaders, the values of the `CullDistance` array are linearly interpolated across each primitive.

If `CullDistance` decorates an input variable, that variable will contain the corresponding value from the `CullDistance` decorated output variable from the previous shader stage.

**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 \texttt{FragCoord} reflect the location of the center of the fragment.

The z component of \texttt{FragCoord} is the interpolated depth value of the primitive.

The w component is the interpolated \( \frac{1}{w} \).

The \texttt{FragCoord} decoration \textbf{must} be used only within fragment shaders.

The variable decorated with \texttt{FragCoord} \textbf{must} be declared using the \texttt{Input} storage class.

The \texttt{Centroid} interpolation decoration is ignored, but allowed, on \texttt{FragCoord}.

The variable decorated with \texttt{FragCoord} \textbf{must} be declared as a four-component vector of 32-bit floating-point values.

\textbf{FragDepth}

To have a shader supply a fragment-depth value, the shader \textbf{must} declare the \texttt{DepthReplacing} execution mode. Such a shader's fragment-depth value will come from the variable decorated with the \texttt{FragDepth} built-in decoration.

This value will be used for any subsequent depth testing performed by the implementation or writes to the depth attachment.

The \texttt{FragDepth} decoration \textbf{must} be used only within fragment shaders.

The variable decorated with \texttt{FragDepth} \textbf{must} be declared using the \texttt{Output} storage class.

The variable decorated with \texttt{FragDepth} \textbf{must} be declared as a scalar 32-bit floating-point value.

\textbf{FrontFacing}

Decorating a variable with the \texttt{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.

The \texttt{FrontFacing} decoration \textbf{must} be used only within fragment shaders.

The variable decorated with \texttt{FrontFacing} \textbf{must} be declared using the \texttt{Input} storage class.

The variable decorated with \texttt{FrontFacing} \textbf{must} be declared as a boolean.

\textbf{GlobalInvocationId}

Decorating a variable with the \texttt{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 \texttt{LocalInvocationId}.

The \texttt{GlobalInvocationId} decoration \textbf{must} be used only within compute shaders.

The variable decorated with \texttt{GlobalInvocationId} \textbf{must} be declared using the \texttt{Input} storage class.
The variable decorated with `GlobalInvocationId` must be declared as a three-component vector of 32-bit integers.

**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.

The `HelperInvocation` decoration must be used only within fragment shaders.

The variable decorated with `HelperInvocation` must be declared using the `Input` storage class.

The variable decorated with `HelperInvocation` must be declared as a boolean.

---

**Note**

It is very likely that a helper invocation will have a value of `SampleMask` fragment shader input value that is zero.

**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.

The `InvocationId` decoration must be used only within tessellation control and geometry shaders.

The variable decorated with `InvocationId` must be declared using the `Input` storage class.

The variable decorated with `InvocationId` must be declared as a scalar 32-bit integer.

**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`.

The `InstanceIndex` decoration must be used only within vertex shaders.

The variable decorated with `InstanceIndex` must be declared using the `Input` storage class.

The variable decorated with `InstanceIndex` must be declared as a scalar 32-bit integer.
### 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 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.

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.

The Layer decoration must be used only within geometry, and fragment shaders.

In a geometry shader, any variable decorated with Layer must be declared using the Output storage class.

In a fragment shader, a variable decorated with Layer contains the layer index of the primitive that the fragment invocation belongs to.

In a fragment shader, any variable decorated with Layer must be declared using the Input storage class.

Any variable decorated with Layer must be declared as a scalar 32-bit integer.

### LocalInvocationId

Decorating a variable with the LocalInvocationId built-in decoration will make that variable contain the location of the current compute shader invocation within the local workgroup. Each component ranges from zero through to the size of the workgroup in that dimension minus one.

The LocalInvocationId decoration must be used only within compute shaders.

The variable decorated with LocalInvocationId must be declared using the Input storage class.

The variable decorated with LocalInvocationId must be declared as a three-component vector of 32-bit integers.

---

**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.

---

### 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:
LocalInvocationIndex =
    LocalInvocationId.z * WorkgroupSize.x * WorkgroupSize.y +
    LocalInvocationId.y * WorkgroupSize.x +
    LocalInvocationId.x;

The **LocalInvocationIndex** decoration **must** be used only within compute shaders.

The variable decorated with **LocalInvocationIndex** **must** be declared using the **Input** storage class.

The variable decorated with **LocalInvocationIndex** **must** be declared as a scalar 32-bit integer.

**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.

The **NumWorkgroups** decoration **must** be used only within compute shaders.

The variable decorated with **NumWorkgroups** **must** be declared using the **Input** storage class.

The variable decorated with **NumWorkgroups** **must** be declared as a three-component vector of 32-bit integers.

**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. A single tessellation control or tessellation evaluation shader **can** read patches of differing sizes, so the value of the **PatchVertices** variable **may** differ between patches.

The **PatchVertices** decoration **must** be used only within tessellation control and tessellation evaluation shaders.

The variable decorated with **PatchVertices** **must** be declared using the **Input** storage class.

The variable decorated with **PatchVertices** **must** be declared as a scalar 32-bit integer.

**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.

The **PointCoord** decoration **must** be used only within fragment shaders.

The variable decorated with **PointCoord** **must** be declared using the **Input** storage class.
The variable decorated with **PointCoord** must be declared as two-component vector of 32-bit floating-point values.

**Note**
Depending on how the point is rasterized, **PointCoord** may never reach (0,0) or (1,1).

**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.

The **PointSize** decoration must be used only within vertex, tessellation control, tessellation evaluation, and geometry shaders.

In a vertex shader, any variable decorated with **PointSize** must be declared using the **Output** storage class.

In a tessellation control, tessellation evaluation, or geometry shader, any variable decorated with **PointSize** must be declared using either the **Input** or **Output** storage class.

Any variable decorated with **PointSize** must be declared as a scalar 32-bit floating-point value.

**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.

**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.

The **Position** decoration must be used only within vertex, tessellation control, tessellation evaluation, and geometry shaders.

In a vertex shader, any variable decorated with **Position** must be declared using the **Output** storage class.

In a tessellation control, tessellation evaluation, or geometry shader, any variable decorated with **Position** must not be declared in a storage class other than **Input** or **Output**.

Any variable decorated with **Position** must be declared as a four-component vector of 32-bit floating-point values.
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.

**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 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 PrimitiveId.

In tessellation control and tessellation evaluation shaders, it will contain the index of the patch within the current set of rendering primitives that correspond 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.

If a geometry shader is present and the fragment shader reads from an input variable decorated with PrimitiveId, then the geometry shader must write to an output variable decorated with PrimitiveId in all execution paths.

The PrimitiveId decoration must be used only within fragment, tessellation control, tessellation evaluation, and geometry shaders.

In a tessellation control, or tessellation evaluation shader, any variable decorated with PrimitiveId must be declared using the Input storage class.

In a geometry shader, any variable decorated with PrimitiveId must be declared using either the Input or Output storage class.

In a fragment shader, any variable decorated with PrimitiveId must be declared using the Input storage class, and either the Geometry or Tessellation capability must also be declared.

Any variable decorated with PrimitiveId must be declared as a scalar 32-bit integer.
Note
When the PrimitiveId decoration is applied to an output variable in the geometry shader, the resulting value is seen through the PrimitiveId decorated input variable in the fragment shader.

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.

The SampleId decoration must be used only within fragment shaders.

The variable decorated with SampleId must be declared using the Input storage class.

The variable decorated with SampleId must be declared as a scalar 32-bit integer.

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.

A variable in the Output storage class decorated with SampleMask is an array of integers forming a bit array in a manner similar 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. However, setting sample mask bits to one will never enable samples not covered by the original primitive. 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.

The SampleMask decoration must be used only within fragment shaders.
Any variable decorated with `SampleMask` must be declared using either the Input or Output storage class.

Any variable decorated with `SampleMask` must be declared as an array of 32-bit integers.

**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 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.

+ The `SamplePosition` decoration must be used only within fragment shaders.

+ The variable decorated with `SamplePosition` must be declared using the Input storage class.

+ The variable decorated with `SamplePosition` must be declared as a two-component vector of 32-bit floating-point values.

**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.

The `TessCoord` decoration must be used only within tessellation evaluation shaders.

The variable decorated with `TessCoord` must be declared using the Input storage class.

The variable decorated with `TessCoord` must be declared as 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.

The `TessLevelOuter` decoration must be used only within tessellation control and tessellation evaluation shaders.

In a tessellation control shader, any variable decorated with `TessLevelOuter` must be declared using the Output storage class.

In a tessellation evaluation shader, any variable decorated with `TessLevelOuter` must be
declared using the Input storage class.

Any 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.

The TessLevelInner decoration must be used only within tessellation control and tessellation evaluation shaders.

In a tessellation control shader, any variable decorated with TessLevelInner must be declared using the Output storage class.

In a tessellation evaluation shader, any variable decorated with TessLevelInner must be declared using the Input storage class.

Any 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.

The VertexIndex decoration must be used only within vertex shaders.

The variable decorated with VertexIndex must be declared using the Input storage class.

The variable decorated with VertexIndex must be declared as a scalar 32-bit integer.

---

**Note**

VertexIndex starts at the same starting value for each instance.

**ViewportIndex**

Decorating a variable with the ViewportIndex built-in decoration will make that variable contain the index of the viewport.
In a 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 and scissor rectangle.

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.

The `ViewportIndex` decoration must be used only within geometry, and fragment shaders.

In a geometry shader, any variable decorated with `ViewportIndex` must be declared using the `Output` storage class.

In a fragment shader, the variable decorated with `ViewportIndex` contains the viewport index of the primitive that the fragment invocation belongs to.

In a fragment shader, any variable decorated with `ViewportIndex` must be declared using the `Input` storage class.

Any variable decorated with `ViewportIndex` must be declared as a scalar 32-bit integer.

**WorkgroupId**

Decorating a variable with the `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.

The `WorkgroupId` decoration must be used only within compute shaders.

The variable decorated with `WorkgroupId` must be declared using the `Input` storage class.

The variable decorated with `WorkgroupId` must be declared as a three-component vector of 32-bit integers.

**WorkgroupSize**

Decorating an object with the `WorkgroupSize` built-in decoration will make that object contain the dimensions of a local workgroup. If an object is decorated with the `WorkgroupSize` decoration, this must take precedence over any execution mode set for `LocalSize`.

The `WorkgroupSize` decoration must be used only within compute shaders.

The object decorated with `WorkgroupSize` must be a specialization constant or a constant.

The object decorated with `WorkgroupSize` must be declared as a three-component vector of 32-bit integers.
Chapter 15. Image Operations

15.1. Image Operations Overview

Vulkan Image Operations are operations performed by those SPIR-V Image Instructions which take an \texttt{OpTypeImage} (representing a \texttt{VkImageView}) or \texttt{OpTypeSampledImage} (representing a \texttt{(VkImageView, VkSampler)} pair) and texel coordinates as operands, and return a value based on one or more neighboring texture elements (\textit{texels}) in the image.

\begin{quote}
\textbf{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.
\end{quote}

Image Operations include the functionality of the following SPIR-V Image Instructions:

- \texttt{OpImageSample*} and \texttt{OpImageSparseSample*} read one or more neighboring texels of the image, and filter the texel values based on the state of the sampler.
  - Instructions with \texttt{ImplicitLod} in the name determine the LOD used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with \texttt{ExplicitLod} in the name determine the LOD used in the sampling operation based on additional coordinates.
  - Instructions with \texttt{Proj} in the name apply homogeneous projection to the coordinates.

- \texttt{OpImageFetch} and \texttt{OpImageSparseFetch} return a single texel of the image. No sampler is used.

- \texttt{OpImageGather} and \texttt{OpImageSparseGather} read neighboring texels and return a single component of each.

- \texttt{OpImageRead} (and \texttt{OpImageSparseRead}) and \texttt{OpImageWrite} read and write, respectively, a texel in the image. No sampler is used.

- Instructions with \texttt{Dref} in the name apply depth comparison on the texel values.

- Instructions with \texttt{Sparse} in the name additionally return a sparse residency code.

15.1.1. Texel Coordinate Systems

Images are addressed by \textit{texel coordinates}. There are three \textit{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 \texttt{OpImageFetch}, \texttt{OpImageSparseFetch}, \texttt{OpImageRead}, \texttt{OpImageSparseRead}, and \texttt{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 \((i, [j], [k], [l])\), 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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{texel_coordinate_systems.png}
\caption{Texel Coordinate Systems, Linear Filtering}
\end{figure}

The Texel Coordinate Systems - For the example shown of an 8×4 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 \(i_1j_1'\).
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 \( ij' \).

## 15.2. Conversion Formulas

### 15.2.1. RGB to Shared Exponent Conversion

An RGB color (red, green, blue) is transformed to a shared exponent color (\( \text{red}_{\text{shared}} \), \( \text{green}_{\text{shared}} \), \( \text{blue}_{\text{shared}} \), \( \text{exp}_{\text{shared}} \)) as follows:

First, the components (red, green, blue) are clamped to (\( \text{red}_{\text{clamped}} \), \( \text{green}_{\text{clamped}} \), \( \text{blue}_{\text{clamped}} \)) as:

\[
\text{red}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{red}))
\]

\[
\text{green}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{green}))
\]

\[
\text{blue}_{\text{clamped}} = \max(0, \min(\text{sharedexp}_{\text{max}}, \text{blue}))
\]

where:
\( N = 9 \)  
\( B = 15 \)  
\( E_{\text{max}} = 31 \)

number of mantissa bits per component  
exponent bias  
maximum possible biased exponent value

\[
\text{sharedexp}_{\text{max}} = \frac{(2^N - 1)}{2^N} \times 2^{(E_{\text{max}} - B)}
\]

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:

\[
\text{max}_{\text{shared}} = \left\lfloor \frac{\text{max}_{\text{clamped}}}{2^{\lfloor \exp' - B - N \rfloor}} + \frac{1}{2} \right\rfloor
\]

\[
\exp_{\text{shared}} = \begin{cases} 
\exp' & \text{for } 0 \leq \text{max}_{\text{shared}} < 2^N \\
\exp' + 1 & \text{for } \text{max}_{\text{shared}} = 2^N 
\end{cases}
\]

Finally, three integer values in the range 0 to \( 2^N \) are computed:

\[
\text{red}_{\text{shared}} = \left\lfloor \frac{\text{red}_{\text{clamped}}}{2^{\lfloor \exp_{\text{shared}} - B - N \rfloor}} + \frac{1}{2} \right\rfloor
\]

\[
\text{green}_{\text{shared}} = \left\lfloor \frac{\text{green}_{\text{clamped}}}{2^{\lfloor \exp_{\text{shared}} - B - N \rfloor}} + \frac{1}{2} \right\rfloor
\]

\[
\text{blue}_{\text{shared}} = \left\lfloor \frac{\text{blue}_{\text{clamped}}}{2^{\lfloor \exp_{\text{shared}} - B - N \rfloor}} + \frac{1}{2} \right\rfloor
\]

15.2.2. 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 \( (\text{red}, \text{green}, \text{blue}) \) as follows:

\[
\text{red} = \text{red}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}
\]

\[
\text{green} = \text{green}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}
\]

\[
\text{blue} = \text{blue}_{\text{shared}} \times 2^{(\exp_{\text{shared}} - B - N)}
\]
where:

\[ N = 9 \text{ (number of mantissa bits per component)} \]

\[ B = 15 \text{ (exponent bias)} \]

### 15.3. 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
- **Format conversion**
- **Texel replacement**
- **Depth comparison**
- **Conversion to RGBA**
- **Component swizzle**

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.

#### 15.3.1. 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. There are a number of 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 `VK_COMPONENT_SWIZZLE_IDENTITY`.

• The `VkImageLayout` of any subresource in the image view does not match that specified in `VkDescriptorImageInfo::imageLayout` used to write the image descriptor.

• If the instruction is `OpImageRead` or `OpImageSparseRead` and the `shaderStorageImageReadWithoutFormat` feature is not enabled, or the instruction is `OpImageWrite` and the `shaderStorageImageWriteWithoutFormat` feature is not enabled, then the SPIR-V Image Format must be compatible with the image view's `format`.

• The sampler `unnormalizedCoordinates` is `VK_TRUE` and any of the limitations of unnormalized coordinates are violated.

• 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.
  ◦ If the image was created with `VkImageCreateInfo::samples` not equal to `VK_SAMPLE_COUNT_1_BIT`, the instruction must have MS = 1.

**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 \textit{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 \textit{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 \textit{Cube map edge handling} is performed.

\textbf{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 \texttt{VK\_FILTER\_NEAREST} filtering within a mip level, since \texttt{VK\_FILTER\_NEAREST} is treated as using \texttt{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 \textbf{should} be replaced by the average of the three values of the adjacent texels in each incident face. However, implementations \textbf{may} replace the cube map corner texel by other methods. The methods are subject to the constraint that if the three available texels have the same value, the resulting filtered texel \textbf{must} have that value.

\textbf{Sparse Validation}

If the texel reads from an unbound region of a sparse image, the texel is a \textit{sparse unbound texel}, and processing continues with \textit{texel replacement}.

\textbf{15.3.2. Format Conversion}

Texels undergo a format conversion from the \texttt{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 \texttt{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.

### 15.3.3. 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</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>
</tbody>
</table>

**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>Table 16. Border Texel Components After Replacement</th>
</tr>
</thead>
</table>
### Texel Aspect or Format

<table>
<thead>
<tr>
<th>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</td>
<td>$\text{Color}_r = B_r$</td>
</tr>
<tr>
<td>Two component color</td>
<td>$[\text{Color}_r, \text{Color}_g] = [B_r, B_g]$</td>
</tr>
<tr>
<td>Three component color</td>
<td>$[\text{Color}_r, \text{Color}_g, \text{Color}_b] = [B_r, B_g, B_b]$</td>
</tr>
<tr>
<td>Four component color</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 `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.

### 15.3.4. 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.

\[
\begin{align*}
D &= 1.0 & & \text{for LEQUAL} \\
D &= Dref \leq D & & \text{for GEQUAL} \\
D &= Dref < D & & \text{for LESS} \\
D &= Dref > D & & \text{for GREATER} \\
D &= Dref = D & & \text{for EQUAL} \\
D &= Dref \neq D & & \text{for NOTEQUAL} \\
true &= \text{for ALWAYS} \\
false &= \text{for NEVER} \quad \text{otherwise}
\end{align*}
\]

where, in the depth comparison:

\[
D_{\text{ref}} = \text{shaderOp.Dref} \quad \text{(from optional SPIR-V operand)}
\]

$D$ (texel depth value)

### 15.3.5. Conversion to RGBA

The texel is expanded from one, two, or three components to four components based on the image base color:
### Table 17. 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.

### 15.3.6. Component Swizzle

All texel input instructions apply a swizzle based on the `VkComponentSwizzle` enums in the `components` member of the `VkImageViewCreateInfo` structure for the image being read.

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:

\[
\text{Color}^{'component'} = \begin{cases} 
\text{Color}_r & \text{for RED swizzle} \\
\text{Color}_g & \text{for GREEN swizzle} \\
\text{Color}_b & \text{for BLUE swizzle} \\
\text{Color}_a & \text{for ALPHA swizzle} \\
0 & \text{for ZERO swizzle} \\
\text{one} & \text{for ONE swizzle} \\
\text{identity} & \text{for IDENTITY swizzle} 
\end{cases}
\]

where:

\[
\text{one} = \begin{cases} 
1.0f & \text{for floating point components} \\
1 & \text{for integer components} 
\end{cases}
\]

\[
\text{identity} = \begin{cases} 
\text{Color}_r & \text{for component} = r \\
\text{Color}_g & \text{for component} = g \\
\text{Color}_b & \text{for component} = b \\
\text{Color}_a & \text{for component} = a 
\end{cases}
\]

If the border color is one of the `VK_BORDER_COLOR_*_OPAQUE_BLACK` enums and the `VkComponentSwizzle` is not `VK_COMPONENT_SWIZZLE_IDENTITY` for all components (or the equivalent identity mapping), the value of the texel after swizzle is undefined.

### 15.3.7. 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.
15.4. 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
  - Coordinate validation
  - Sparse validation
- Texel output format conversion

15.4.1. 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.

**15.4.2. 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.

**15.4.3. 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.

**15.4.4. 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.

15.5. Normalized Texel Coordinate Operations

If the image sampler instruction provides normalized texel coordinates, some of the following operations are performed.

15.5.1. 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:

\[
\begin{align*}
    s &= \frac{s}{q}, & \text{for 1D, 2D, or 3D image} \\
    t &= \frac{t}{q}, & \text{for 2D or 3D image} \\
    r &= \frac{r}{q}, & \text{for 3D image} \\
    D_{\text{ref}} &= \frac{D_{\text{ref}}}{q}, & \text{if provided}
\end{align*}
\]

15.5.2. 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:

\[
\begin{align*}
    \frac{\partial s}{\partial x} &= dPdx(s), & \frac{\partial s}{\partial y} &= dPdy(s), & \text{for 1D, 2D, Cube, or 3D image} \\
    \frac{\partial t}{\partial x} &= dPdx(t), & \frac{\partial t}{\partial y} &= dPdy(t), & \text{for 2D, Cube, or 3D image} \\
    \frac{\partial u}{\partial x} &= dPdx(u), & \frac{\partial u}{\partial y} &= dPdy(u), & \text{for Cube or 3D image}
\end{align*}
\]

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 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.
15.5.3. 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 \((s_{face},t_{face})\). The direction vector is also used to transform the derivatives to per-face derivatives.

15.5.4. 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 18. 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>
<tr>
<td>(-r_z)</td>
<td>5</td>
<td>Negative Z</td>
<td>(-r_x)</td>
<td>(-r_y)</td>
<td>(r_z)</td>
</tr>
</tbody>
</table>

### Table 19. Cube map derivative selection

<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>(\partial s_c / \partial x)</th>
<th>(\partial s_c / \partial y)</th>
<th>(\partial t_c / \partial x)</th>
<th>(\partial t_c / \partial y)</th>
<th>(\partial r_c / \partial x)</th>
<th>(\partial r_c / \partial y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+r_x)</td>
<td>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(+\partial r_x / \partial x)</td>
<td>(+\partial r_x / \partial y)</td>
</tr>
<tr>
<td>(-r_x)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(-\partial r_x / \partial x)</td>
<td>(-\partial r_x / \partial y)</td>
</tr>
<tr>
<td>(+r_y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(+\partial r_x / \partial x)</td>
<td>(+\partial r_x / \partial y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
</tr>
<tr>
<td>(-r_y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(-\partial r_x / \partial x)</td>
<td>(-\partial r_x / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
</tr>
<tr>
<td>(+r_z)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(+\partial r_z / \partial x)</td>
<td>(+\partial r_z / \partial y)</td>
</tr>
<tr>
<td>(-r_z)</td>
<td>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
<td>(-\partial r_y / \partial x)</td>
<td>(-\partial r_y / \partial y)</td>
<td>(-\partial r_z / \partial x)</td>
<td>(-\partial r_z / \partial y)</td>
</tr>
</tbody>
</table>
15.5.5. Cube Map Coordinate Transformation

\[ s_{face} = \frac{1}{2} \times \frac{s_c + \frac{1}{2}}{|r_c|} \]
\[ t_{face} = \frac{1}{2} \times \frac{t_c + \frac{1}{2}}{|r_c|} \]

15.5.6. Cube Map Derivative Transformation

\[ \frac{\partial s_{face}}{\partial x} = \frac{\partial}{\partial x} \left( \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2} \right) \]
\[ \frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \frac{\partial}{\partial x} \left( \frac{s_c}{|r_c|} \right) \]
\[ \frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \left( \frac{|r_c| \times \partial s_c / \partial x - s_c \times \partial r_c / \partial x}{(r_c)^2} \right) \]

15.5.7. 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 implicit LOD selected can be queried using the SPIR-V instruction `OpImageQueryLod`, which gives access to the \( \lambda' \) and \( d_i \) values, defined below. These values must be computed with `mipmapPrecisionBits` of accuracy and may be subject to implementation-specific maxima and minima for very large, out-of-range values.

**Scale Factor Operation**

The magnitude of the derivatives are calculated by:

\[ m_{ux} = |\partial s/\partial x| \times w_{base} \]
\[ m_{vx} = |\partial t/\partial x| \times h_{base} \]
\[ m_{wx} = |\partial r/\partial x| \times d_{base} \]
\[ m_{uy} = |\partial s/\partial y| \times w_{base} \]
\[ m_{vy} = |\partial t/\partial y| \times h_{base} \]
\[ m_{wy} = |\partial r/\partial y| \times d_{\text{base}} \]

where:

\[ \partial t/\partial x = \partial t/\partial y = 0 \text{ (for 1D images)} \]

\[ \partial r/\partial x = \partial r/\partial y = 0 \text{ (for 1D, 2D or Cube images)} \]

and:

\[ w_{\text{base}} = \text{image.w} \]

\[ h_{\text{base}} = \text{image.h} \]

\[ d_{\text{base}} = \text{image.d} \]

(for the \text{baseMipLevel}, from the image descriptor).

A point sampled in screen space has an elliptical footprint in texture space. The minimum and maximum scale factors (\( \rho_{\text{min}}, \rho_{\text{max}} \)) \textbf{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 \( \rho_y \) \textbf{may} be approximated with functions \( f_x \) and \( f_y \), subject to the following constraints:

\[ f_x \text{is continuous and monotonically increasing in each of} m_{ux}, m_{vx}, \text{and} m_{wx} \]

\[ f_y \text{is continuous and monotonically increasing in each of} m_{uy}, m_{vy}, \text{and} m_{wy} \]

\[ \max(|m_{ux}|, |m_{vx}|, |m_{wx}|) \leq f_x \leq \sqrt{2(|m_{ux}| + |m_{vx}| + |m_{wx}|)} \]

\[ \max(|m_{uy}|, |m_{vy}|, |m_{wy}|) \leq f_y \leq \sqrt{2(|m_{uy}| + |m_{vy}| + |m_{wy}|)} \]

The minimum and maximum scale factors (\( \rho_{\text{min}}, \rho_{\text{max}} \)) are determined by:

\[ \rho_{\text{max}} = \max(\rho_x, \rho_y) \]

\[ \rho_{\text{min}} = \min(\rho_x, \rho_y) \]

The ratio of anisotropy is determined by:

\[ \eta = \min(\rho_{\text{max}}/\rho_{\text{min}}, \max_{\text{Aniso}}) \]
where:

\[
\text{sampler.max}_{\text{Aniso}} = \text{maxAnisotropy} \text{ (from sampler descriptor)}
\]

\[
\text{limits.max}_{\text{Aniso}} = \text{maxSamplerAnisotropy} \text{ (from physical device limits)}
\]

\[
\text{max}_{\text{Aniso}} = \min(\text{sampler.max}_{\text{Aniso}}, \text{limits.max}_{\text{Aniso}})
\]

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 \( N \) 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 \( \text{max}_{\text{Aniso}} \). 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 \( \text{VkPhysicalDeviceFeatures}::\text{samplerAnisotropy} \) or \( \text{VkSamplerCreateInfo}::\text{anisotropyEnable} \) are \( \text{VK_FALSE} \), \( \text{max}_{\text{Aniso}} \) 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:

\[
\lambda_{\text{base}}(x, y) = \begin{cases} 
\text{shaderOp.Lod} & \text{(from optional SPIR-V operand)} \\
\log_{\eta}(\frac{\rho_{\text{max}}}{\eta}) & \text{otherwise}
\end{cases}
\]

\[
\lambda'(x, y) = \lambda_{\text{base}} + \text{clamp}(\text{sampler.bias} + \text{shaderOp.bias}, -\text{maxSamplerLodBias}, \text{maxSamplerLodBias})
\]

\[
\lambda = \begin{cases} 
\text{lod}_{\text{max}}, & \lambda' > \text{lod}_{\text{max}} \\
\lambda', & \text{lod}_{\text{min}} \leq \lambda' \leq \text{lod}_{\text{max}} \\
\text{lod}_{\text{min}}, & \lambda' < \text{lod}_{\text{min}} \\
\text{undefined}, & \text{lod}_{\text{min}} > \text{lod}_{\text{max}}
\end{cases}
\]

where:

\[
\text{sampler.bias} = \text{mipLodBias} \quad \text{(from sampler descriptor)}
\]

\[
\text{shaderOp.bias} = \begin{cases} 
\text{Bias} & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{sampler.lod}_{\text{min}} = \text{minLod} \quad \text{(from sampler descriptor)}
\]

\[
\text{shaderOp.lod}_{\text{min}} = \begin{cases} 
\text{minLod} & \text{(from optional SPIR-V operand)} \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{lod}_{\text{min}} = \max(\text{sampler.lod}_{\text{min}}, \text{shaderOp.lod}_{\text{min}}) \\
\text{lod}_{\text{max}} = \text{maxLod} \quad \text{(from sampler descriptor)}
\]
and maxSamplerLodBias is the value of the VkPhysicalDeviceLimits feature maxSamplerLodBias.

### Image Level(s) Selection

The image level(s) \( d, d_{hi}, \) and \( d_{lo} \) which texels are read from are determined by an image-level parameter \( d_l \), which is computed based on the LOD parameter, as follows:

\[
d_l = \begin{cases} 
    \text{nearest}(d'), & \text{mipmapMode is } \text{VK_SAMPLER_MIPMAP_MODE_NEAREST} \\
    d', & \text{otherwise}
\end{cases}
\]

where:

\[
d' = \text{level}_{base} + \text{clamp}(\lambda, 0, q)
\]

\[
\text{nearest}(d') = \begin{cases} 
    \lfloor d' + 0.5 \rfloor - 1, & \text{preferred} \\
    \lfloor d' + 0.5 \rfloor, & \text{alternative}
\end{cases}
\]

and:

\[
\text{level}_{base} = \text{baseMipLevel}
\]

\[
q = \text{levelCount} - 1
\]

\text{baseMipLevel} and \text{levelCount} are taken from the subresourceRange of the image view.

If the sampler’s mipmapMode is \text{VK_SAMPLER_MIPMAP_MODE_NEAREST}, then the level selected is \( d = d_l \).

If the sampler’s mipmapMode is \text{VK_SAMPLER_MIPMAP_MODE_LINEAR}, two neighboring levels are selected:

\[
d_{hi} = \lfloor d_l \rfloor \\
d_{lo} = \min(d_{hi} + 1, q) \\
\delta = d_l - d_{hi}
\]

\( \delta \) is the fractional value, quantized to the number of mipmap precision bits, used for linear filtering between levels.

### 15.5.8. (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 \( d_{lo} \)).

\[
u(x, y) = s(x, y) \times width_{scale} + \Delta_i
\]

\[
v(x, y) = \begin{cases} 
    0, & \text{for } 1D \text{ images} \\
    t(x, y) \times height_{scale} + \Delta_j, & \text{otherwise}
\end{cases}
\]

\[
w(x, y) = \begin{cases} 
    0, & \text{for } 2D \text{ or Cube images} \\
    r(x, y) \times depth_{scale} + \Delta_k, & \text{otherwise}
\end{cases}
\]

\[
a(x, y) = \begin{cases} 
    a(x, y), & \text{for array images} \\
    0, & \text{otherwise}
\end{cases}
\]
where:

\[
\text{width}_{\text{scale}} = \text{width}_{\text{level}}
\]

\[
\text{height}_{\text{scale}} = \text{height}_{\text{level}}
\]

\[
\text{depth}_{\text{scale}} = \text{depth}_{\text{level}}
\]

and where \((\Delta_i, \Delta_j, \Delta_k)\) 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.

### 15.6. Unnormalized Texel Coordinate Operations

#### 15.6.1. \((u,v,w,a)\) 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:

\[
\text{RNE}(a) = \begin{cases} 
\text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\
[a + 0.5] & \text{alternative}
\end{cases}
\]

The sample index \(n\) is assigned the value 0.

Nearest filtering (VK\_FILTER\_NEAREST) computes the integer texel coordinates that the unnormalized coordinates lie within:

\[
i = \lfloor u + \text{shift} \rfloor \\
j = \lfloor v + \text{shift} \rfloor \\
k = \lfloor w + \text{shift} \rfloor
\]

where:

\[
\text{shift} = 0.0
\]

Linear filtering (VK\_FILTER\_LINEAR) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of \(i_0, i_1, j_0, j_1, k_0\) or \(k_1\),
as well as weights \( \alpha \), \( \beta \), and \( \gamma \).

\[
\begin{align*}
    i_0 &= \lfloor u - \text{shift} \rfloor \\
    i_1 &= i_0 + 1 \\
    j_0 &= \lfloor v - \text{shift} \rfloor \\
    j_1 &= j_0 + 1 \\
    k_0 &= \lfloor w - \text{shift} \rfloor \quad = k_0 + 1
\end{align*}
\]

\[
\begin{align*}
    \alpha &= \text{frac}(u - \text{shift}) \\
    \beta &= \text{frac}(v - \text{shift}) \\
    \gamma &= \text{frac}(w - \text{shift})
\end{align*}
\]

where:

\[
\text{shift} = 0.5
\]

and where:

\[
\text{frac}(x) = x - \lfloor x \rfloor
\]

where the number of fraction bits retained is specified by \( \text{VkPhysicalDeviceLimits} :: \text{subTexelPrecisionBits} \).

### 15.7. Integer Texel Coordinate Operations

The \text{OpImageFetch} and \text{OpImageFetchSparse} SPIR-V instructions \textbf{may} supply a LOD from which texels are to be fetched using the optional SPIR-V operand \text{Lod}. Other integer-coordinate operations \textbf{must} not. If the \text{Lod} is provided then it \textbf{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 undefined, and any writes (if supported) are discarded.

### 15.8. Image Sample Operations

#### 15.8.1. Wrapping Operation

\textit{Cube} images ignore the wrap modes specified in the sampler. Instead, if \text{VK\_FILTER\_NEAREST} is used within a mip level then \text{VK\_SAMPLER\_ADDRESS\_MODE\_CLAMP\_TO\_EDGE} is used, and if \text{VK\_FILTER\_LINEAR} is used within a mip level then sampling at the edges is performed as described earlier in the \textit{Cube map edge handling} section.

The first integer texel coordinate \( i \) is transformed based on the \textit{addressModeU} parameter of the sampler.
where:

\[
mirror(n) = \begin{cases} 
  n & \text{for } n \geq 0 \\
  -(1+n) & \text{otherwise}
\end{cases}
\]

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.

### 15.8.2. 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 operation does not use the `ConstOffsets` image operand then the four texels form the 2 × 2 rectangle used for texture filtering:

\[
\begin{align*}
\tau[R] &= \tau_{\{0,0\}}[\text{level}_{\text{base}}][\text{comp}] \\
\tau[G] &= \tau_{\{0,1\}}[\text{level}_{\text{base}}][\text{comp}] \\
\tau[B] &= \tau_{\{1,0\}}[\text{level}_{\text{base}}][\text{comp}] \\
\tau[A] &= \tau_{\{1,1\}}[\text{level}_{\text{base}}][\text{comp}]
\end{align*}
\]

If the operation does use the `ConstOffsets` image operand then the offsets allow a custom filter to be defined:

\[
\begin{align*}
\tau[R] &= \tau_{\{0,0\}+\Delta_0}[\text{level}_{\text{base}}][\text{comp}] \\
\tau[G] &= \tau_{\{0,0\}+\Delta_1}[\text{level}_{\text{base}}][\text{comp}] \\
\tau[B] &= \tau_{\{0,0\}+\Delta_2}[\text{level}_{\text{base}}][\text{comp}] \\
\tau[A] &= \tau_{\{0,0\}+\Delta_3}[\text{level}_{\text{base}}][\text{comp}]
\end{align*}
\]

where:

\[
\tau[\text{level}_{\text{base}}][\text{comp}] = \begin{cases} 
  \tau[\text{level}_{\text{base}}][R], & \text{for } \text{comp} = 0 \\
  \tau[\text{level}_{\text{base}}][G], & \text{for } \text{comp} = 1 \\
  \tau[\text{level}_{\text{base}}][B], & \text{for } \text{comp} = 2 \\
  \tau[\text{level}_{\text{base}}][A], & \text{for } \text{comp} = 3
\end{cases}
\]  

\text{comp} \text{ from SPIR-V operand Component}

### 15.8.3. 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, \texttt{VK\_FILTER\_NEAREST} filtering selects a single value using the \((i, j, k)\) texel coordinates, with all texels taken from layer \(l\).

\[
\tau_{\text{level}} = \begin{cases} 
\tau_{i,j,k}[\text{level}], & \text{for 3D image} \\
\tau_{i,j}[\text{level}], & \text{for 2D or Cube image} \\
\tau_{i}[\text{level}], & \text{for 1D image}
\end{cases}
\]

Texel Linear Filtering

Within a mip level, \texttt{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:

\[
\begin{align*}
w_{i_0} &= (1 - \alpha) \\
w_{i_1} &= \alpha \\
w_{j_0} &= (1 - \beta) \\
w_{j_1} &= \beta \\
w_{k_0} &= (1 - \gamma) \\
w_{k_1} &= \gamma
\end{align*}
\]

The values of multiple texels, together with their weights, are combined using a weighted average to produce a filtered value:

\[
\begin{align*}
\tau_{3D} &= \sum_{k = k_0}^{k_1} \sum_{j = j_0}^{j_1} \sum_{i = i_0}^{i_1} (w_i) (w_j) (w_k) (w_{ij}) \\
\tau_{2D} &= \sum_{j = j_0}^{j_1} \sum_{i = i_0}^{i_1} (w_i) (w_j) \\
\tau_{1D} &= \sum_{i = i_0}^{i_1} (w_i)
\end{align*}
\]

Texel Mipmap Filtering

\texttt{VK\_SAMPLER\_MIPMAP\_MODE\_NEAREST} filtering returns the value of a single mipmap level,

\[
\tau = \tau[d].
\]

\texttt{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:

\[
\begin{align*}
w_{hi} &= (1 - \delta) \\
w_{lo} &= \delta
\end{align*}
\]

The values of multiple mipmap levels together with their linear weights, are combined using a weighted average to produce a final filtered value:

\[
\tau = (w_{hi}) \tau[hi] + (w_{lo}) \tau[lo]
\]
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 be 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.

\[
\tau_{2D_{\text{Aniso}}} = \frac{1}{N} \sum_{i=1}^{N} \tau_{2D}\left( u\left( x - \frac{1}{2} + \frac{i}{N+1}, y \right), \left( v\left( x - \frac{1}{2} + \frac{i}{N+1}, y \right) \right) \right), \quad \text{when } \rho_x > \rho_y
\]

\[
\tau_{2D_{\text{Aniso}}} = \frac{1}{N} \sum_{i=1}^{N} \tau_{2D}\left( u\left( x, y - \frac{1}{2} + \frac{i}{N+1} \right), \left( v\left( x, y - \frac{1}{2} + \frac{i}{N+1} \right) \right) \right), \quad \text{when } \rho_y \geq \rho_x
\]

15.9. 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.
Chapter 16. 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, and Timestamp Queries.

16.1. 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:

```
VK_DEFINE_NON_DISPATCHABLE_HANDLE(VkQueryPool)
```

To create a query pool, call:

```
VkResult vkCreateQueryPool(
    VkDevice device,                        // device
    const VkQueryPoolCreateInfo* pCreateInfo, // pCreateInfo
    const VkAllocationCallbacks* pAllocator,   // pAllocator
    VkQueryPool* pQueryPool);                // 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)

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a valid pointer to a valid VkQueryPoolCreateInfo structure
- If `pAllocator` is not NULL, `pAllocator` must be a valid pointer to a valid VkAllocationCallbacks structure
- `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
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 an extension-specific 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**

- If the pipeline statistics queries feature is not enabled, `queryType` must not be `VK_QUERY_TYPE_PIPELINE_STATISTICS`
- If `queryType` is `VK_QUERY_TYPE_PIPELINE_STATISTICS`, `pipelineStatistics` must be a valid combination of `VkQueryPipelineStatisticFlagBits` values
- **queryCount** must be greater than 0
Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- `queryType` must be a valid `VkQueryType` value

```c
typedef VkFlags VkQueryPoolCreateFlags;
```

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

To destroy a query pool, call:

```c
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

- All submitted commands that refer to `queryPool` must have completed execution
- If `VkAllocationCallbacks` were provided when `queryPool` was created, a compatible set of callbacks must be provided here
- If no `VkAllocationCallbacks` were provided when `queryPool` was created, `pAllocator` must be `NULL`

Valid Usage (Implicit)

- `device` must be a valid `VkDevice` handle
- If `queryPool` is not `VK_NULL_HANDLE`, `queryPool` must be a valid `VkQueryPool` handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a valid pointer to a valid `VkAllocationCallbacks` structure
- 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:

```c
typedef enum VkQueryType {
    VK_QUERY_TYPE_OCCLUSION = 0,
    VK_QUERY_TYPE_PIPELINE_STATISTICS = 1,
    VK_QUERY_TYPE_TIMESTAMP = 2,
} VkQueryType;
```

- `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.

16.2. Query Operation

The operation of queries is controlled by the commands `vkCmdBeginQuery`, `vkCmdEndQuery`, `vkCmdResetQueryPool`, `vkCmdCopyQueryPoolResults`, 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` 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 command 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.

To reset a range of queries in a query pool on a queue, call:
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 \([firstQuery, firstQuery + queryCount - 1]\) to unavailable.

### Valid Usage

- **firstQuery** must be less than the number of queries in **queryPool**
- The sum of **firstQuery** and **queryCount** must be less than or equal to the number of queries in **queryPool**
- All queries used by the command must not be active

### Valid Usage (Implicit)

- **commandBuffer** must be a valid **VkCommandBuffer** handle
- **queryPool** must be a valid **VkQueryPool** handle
- **commandBuffer** must be in the recording state
- The **VkCommandPool** that **commandBuffer** was allocated from must support graphics, or compute operations
- This command must only be called outside of a render pass instance
- 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
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.

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).

To begin a query, call:

```c
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.

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

- `queryPool` must have been created with a `queryType` that differs from that of any queries that are active within `commandBuffer`
- All queries used by the command must be unavailable
- The `queryType` used to create `queryPool` must not be `VK_QUERY_TYPE_TIMESTAMP`
- 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`
- `query` must be less than the number of queries in `queryPool`
- If the `queryType` used to create `queryPool` was `VK_QUERY_TYPE_OCCLUSION`, the `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- 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
- 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

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `queryPool` must be a valid `VkQueryPool` handle
- `flags` must be a valid combination of `VkQueryControlFlagBits` values
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations
- 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>Graphics</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td>Compute</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
typedef enum VkQueryControlFlagBits {
    VK_QUERY_CONTROL_PRECISE_BIT = 0x00000001,
} VkQueryControlFlagBits;
```

- `VK_QUERY_CONTROL_PRECISE_BIT` specifies the precision of occlusion queries.

```c
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:

```c
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.

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

- All queries used by the command must be active
- query must be less than the number of queries in queryPool

Valid Usage (Implicit)

- commandBuffer must be a valid VkCommandBuffer handle
- queryPool must be a valid VkQueryPool handle
- commandBuffer must be in the recording state
- The VkCommandPool that commandBuffer was allocated from must support graphics, or compute operations
- 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>

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, 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.

- 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.
• 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.

• 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
VkResult vkGetQueryPoolResults(
    VkDevice                                    device,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount,
    size_t                                      dataSize,
    void*                                       pData,
    VkDeviceSize                                stride,
    VkQueryResultFlags                          flags);
```

• `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 \texttt{VK_QUERY_RESULT_WAIT_BIT} and \texttt{VK_QUERY_RESULT_PARTIAL_BIT} are both not set then no result values are written to \texttt{pData} for queries that are in the unavailable state at the time of the call, and \texttt{vkGetQueryPoolResults} returns \texttt{VK_NOT_READY}. However, availability state is still written to \texttt{pData} for those queries if \texttt{VK_QUERY_RESULT_WITH_AVAILABILITY_BIT} is set.

\textit{Note}
Applications \textbf{must} take care to ensure that use of the \texttt{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 \texttt{vkCmdResetQueryPool}, \texttt{vkCmdBeginQuery}, and \texttt{vkCmdEndQuery} for that query, then the query will remain in the available state until the \texttt{vkCmdResetQueryPool} command executes on a queue. Applications \textbf{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 \texttt{VK_QUERY_RESULT_WAIT_BIT} is used in combination with \texttt{VK_QUERY_RESULT_WITH_AVAILABILITY_BIT}. In this case, the returned availability status \textbf{may} reflect the result of a previous use of the query unless the \texttt{vkCmdResetQueryPool} command has been executed since the last use of the query.

\textit{Note}
Applications \textbf{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 \texttt{VK_QUERY_RESULT_PARTIAL_BIT} is set, \texttt{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 \texttt{pData} for that query.

If \texttt{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 \texttt{VK_QUERY_RESULT_WITH_AVAILABILITY_BIT} is used, implementations \textbf{must} guarantee that if they return a non-zero availability value then the numerical results \textbf{must} be valid, assuming the results are not reset by a subsequent command.

\textit{Note}
Satisfying this guarantee \textbf{may} require careful ordering by the application, e.g. to read the availability status before reading the results.
Valid Usage

- **firstQuery** must be less than the number of queries in **queryPool**
- If **VK_QUERY_RESULT_64_BIT** is not set in **flags**, then **pData** and **stride** must be multiples of 4
- If **VK_QUERY_RESULT_64_BIT** is set in **flags**, then **pData** and **stride** must be multiples of 8
- The sum of **firstQuery** and **queryCount** must be less than or equal to the number of queries in **queryPool**
- **dataSize** must be large enough to contain the result of each query, as described here
- If the **queryType** used to create **queryPool** was **VK_QUERY_TYPE_TIMESTAMP**, **flags** must not contain **VK_QUERY_RESULT_PARTIAL_BIT**

Valid Usage (Implicit)

- **device** must be a valid **VkDevice** handle
- **queryPool** must be a valid **VkQueryPool** handle
- **pData** must be a valid pointer to an array of **dataSize** bytes
- **flags** must be a valid combination of **VkQueryResultFlagBits** values
- **dataSize** must be greater than 0
- **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:
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,
} 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.

typedef VkFlags VkQueryResultFlags;

**VkQueryResultFlags** is a bitmask type for setting a mask of zero or more **VkQueryResultFlagBits**.

To copy query statuses and numerical results directly to buffer memory, call:

```c
void vkCmdCopyQueryPoolResults(
    VkCommandBuffer commandBuffer,          // commandBuffer is the command buffer into which this command will be recorded.
    VkQueryPool queryPool,                  // queryPool is the query pool managing the queries containing the desired results.
    uint32_t firstQuery,                    // firstQuery is the initial query index.
    uint32_t queryCount,                    // queryCount is the number of queries. firstQuery and queryCount together define a range of queries.
    VkBuffer dstBuffer,                     // dstBuffer is a VkBuffer object that will receive the results of the copy command.
    VkDeviceSize dstOffset,                 // dstOffset is an offset into dstBuffer.
    VkDeviceSize stride,                    // 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.
    VkQueryResultFlags flags);              // 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, 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.

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

- \texttt{dstOffset} \textbf{must} be less than the size of \texttt{dstBuffer}
- \texttt{firstQuery} \textbf{must} be less than the number of queries in \texttt{queryPool}
- The sum of \texttt{firstQuery} and \texttt{queryCount} \textbf{must} be less than or equal to the number of queries in \texttt{queryPool}
- If \texttt{VK_QUERY_RESULT_64_BIT} is not set in \texttt{flags} then \texttt{dstOffset} and \texttt{stride} \textbf{must} be multiples of 4
- If \texttt{VK_QUERY_RESULT_64_BIT} is set in \texttt{flags} then \texttt{dstOffset} and \texttt{stride} \textbf{must} be multiples of 8
- \texttt{dstBuffer} \textbf{must} have enough storage, from \texttt{dstOffset}, to contain the result of each query, as described \texttt{here}
- \texttt{dstBuffer} \textbf{must} have been created with \texttt{VK_BUFFER_USAGE_TRANSFER_DST_BIT} usage flag
- If \texttt{dstBuffer} is non-sparse then it \textbf{must} be bound completely and contiguously to a single \texttt{VkDeviceMemory} object
- If the \texttt{queryType} used to create \texttt{queryPool} was \texttt{VK_QUERY_TYPE_TIMESTAMP}, \texttt{flags} \textbf{must} not contain \texttt{VK_QUERY_RESULT_PARTIAL_BIT}

Valid Usage (Implicit)

- \texttt{commandBuffer} \textbf{must} be a valid \texttt{VkCommandBuffer} handle
- \texttt{queryPool} \textbf{must} be a valid \texttt{VkQueryPool} handle
- \texttt{dstBuffer} \textbf{must} be a valid \texttt{VkBuffer} handle
- \texttt{flags} \textbf{must} be a valid combination of \texttt{VkQueryResultFlagBits} values
- \texttt{commandBuffer} \textbf{must} be in the \texttt{recording state}
- The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from \textbf{must} support graphics, or compute operations
- This command \textbf{must} only be called outside of a render pass instance
- Each of \texttt{commandBuffer}, \texttt{dstBuffer}, and \texttt{queryPool} \textbf{must} have been created, allocated, or retrieved from the same \texttt{VkDevice}

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
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.

### 16.3. 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.

### 16.4. 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
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
typedef VkFlags VkQueryPipelineStatisticFlags;
```

`VkQueryPipelineStatisticFlags` is a bitmask type for setting a mask of zero or more `VkQueryPipelineStatisticFlagBits`.

### 16.5. 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:

```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.

`vkCmdWriteTimestamp` 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.

`vkCmdCopyQueryPoolResults` can then be called to copy the timestamp value from the query pool into buffer memory, with ordering and synchronization behavior equivalent to how other queries operate. Timestamp values can also be retrieved from the query pool using `vkGetQueryPoolResults`. As with other queries, the query must be reset using `vkCmdResetQueryPool` before requesting the timestamp value be written to it.

While `vkCmdWriteTimestamp` can be called inside or outside of a render pass instance, `vkCmdCopyQueryPoolResults` must only be called outside of a render pass instance.

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.

Valid Usage

- `queryPool` must have been created with a `queryType` of `VK_QUERY_TYPE_TIMESTAMP`
- The query identified by `queryPool` and `query` must be unavailable
- The command pool's queue family must support a non-zero `timestampValidBits`

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `pipelineStage` must be a valid `VkPipelineStageFlagBits` value
- `queryPool` must be a valid `VkQueryPool` handle
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- 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>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>
Chapter 17. Clear Commands

17.1. 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
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_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

- **image** must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- If **image** is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object
- **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`
- **imageLayout** must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`
- The `VkImageSubresourceRange::aspectMask` members of the elements of the `pRanges` array **must** each only include `VK_IMAGE_ASPECT_COLOR_BIT`
- 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
- 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
- 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
- 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
- **image** must not have a compressed or depth/stencil format
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `image` must be a valid `VkImage` handle
- `imageLayout` must be a valid `VkImageLayout` value
- `pColor` must be a valid pointer to a valid `VkClearColorValue` union
- `pRanges` must be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics, or compute operations
- This command must only be called outside of a render pass instance
- `rangeCount` must be greater than 0
- 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>Compute</td>
<td></td>
</tr>
</tbody>
</table>

To clear one or more subranges of a depth/stencil image, call:

```c
void vkCmdClearDepthStencilImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearColorValue*                   pDepthStencil,
    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_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

• **image** must have been created with **VK_IMAGE_USAGE_TRANSFER_DST_BIT** usage flag

• If **image** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object

• **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**

• **imageLayout** must be either of **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**

• 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**

• 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

• 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

• 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

• 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

• 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

• 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

• **image** must have a depth/stencil format
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `image` must be a valid `VkImage` handle
- `imageLayout` must be a valid `VkImageLayout` value
- `pDepthStencil` must be a valid pointer to a valid `VkClearDepthStencilValue` structure
- `pRanges` must be a valid pointer to an array of `rangeCount` valid `VkImageSubresourceRange` structures
- `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
- `rangeCount` must be greater than 0
- 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 Secondary</td>
<td>Outside</td>
<td>Graphics</td>
<td>Transfer</td>
</tr>
</tbody>
</table>

Clears outside render pass instances are treated as transfer operations for the purposes of memory barriers.

17.2. 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:
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.

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

• 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

• 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

• 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

• The rect member of each element of pRects must have an extent.width greater than 0

• The rect member of each element of pRects must have an extent.height greater than 0

• The rectangular region specified by each element of pRects must be contained within the render area of the current render pass instance

• The layers specified by each element of pRects must be contained within every attachment that pAttachments refers to

• The layerCount member of each element of pRects must not be 0

Valid Usage (Implicit)

• commandBuffer must be a valid VkCommandBuffer handle

• pAttachments must be a valid pointer to an array of attachmentCount valid VkClearAttachment structures

• pRects must be a valid pointer to an array of rectCount VkClearRect structures

• commandBuffer must be in the recording state

• The VkCommandPool that commandBuffer was allocated from must support graphics operations

• This command must only be called inside of a render pass instance

• attachmentCount must be greater than 0

• 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
The **VkClearRect** structure is defined as:

```c
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 \([\text{baseArrayLayer}, \text{baseArrayLayer} + \text{layerCount}]\) counting from the base layer of the attachment image view are cleared.

The **VkClearAttachment** structure is defined as:

```c
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 \(\text{VK_IMAGE_ASPECT_COLOR_BIT}\) is set in **aspectMask**, in which case it is an index to the \(\text{pColorAttachments}\) 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

- If aspectMask includes VK_IMAGE_ASPECT_COLOR_BIT, it must not include VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT
- aspectMask must not include VK_IMAGE_ASPECT_METADATA_BIT
- clearValue must be a valid VkClearValue union

### Valid Usage (Implicit)

- aspectMask must be a valid combination of VkImageAspectFlagBits values
- aspectMask must not be 0

### 17.3. Clear Values

The VkClearColorValue structure is defined as:

```c
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
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

- `depth` must be between 0.0 and 1.0, inclusive

The `VkClearValue` union is defined as:

```c
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.

## 17.4. Filling Buffers

To clear buffer data, call:

```c
void vkCmdFillBuffer(
    VkCommandBuffer commandBuffer,  // commandBuffer is the command buffer into which the command will be recorded.
    VkBuffer dstBuffer,             // dstBuffer
    VkDeviceSize dstOffset,         // dstOffset
    VkDeviceSize size,              // size
    uint32_t data);                 // data
```

• **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

- **dstOffset** **must** be less than the size of **dstBuffer**
- **dstOffset** **must** be a multiple of 4
- If **size** is not equal to **VK_WHOLE_SIZE**, **size** **must** be greater than 0
- If **size** is not equal to **VK_WHOLE_SIZE**, **size** **must** be less than or equal to the size of **dstBuffer minus dstOffset**
- If **size** is not equal to **VK_WHOLE_SIZE**, **size** **must** be a multiple of 4
- **dstBuffer** **must** have been created with **VK_BUFFER_USAGE_TRANSFER_DST_BIT** usage flag
- The **VkCommandPool** that **commandBuffer** was allocated from **must** support graphics or compute operations
- If **dstBuffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object

### Valid Usage (Implicit)

- **commandBuffer** **must** be a valid **VkCommandBuffer** handle
- **dstBuffer** **must** be a valid **VkBuffer** handle
- **commandBuffer** **must** be in the **recording state**
- The **VkCommandPool** that **commandBuffer** was allocated from **must** support graphics or compute operations
- This command **must** only be called outside of a render pass instance
- 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>Graphics</td>
<td>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td>Compute</td>
<td></td>
</tr>
</tbody>
</table>

17.5. 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

- **`dstOffset` must** be less than the size of `dstBuffer`
- **dataSize must** be less than or equal to the size of `dstBuffer` minus `dstOffset`
- **`dstBuffer` must** have been created with `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
- **dataSize must** be less than or equal to 65536
- **dataSize must** be a multiple of 4
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `dstBuffer` must be a valid `VkBuffer` handle
- `pData` must be a valid pointer to an array of `dataSize` bytes
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- This command must only be called outside of a render pass instance
- `dataSize` must be greater than 0
- 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 18. 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 `vkCmdCopyBuffer` and a portion of an image can be copied to another image with `vkCmdCopyImage`. Image data can also be copied to and from buffer memory using `vkCmdCopyImageToBuffer` and `vkCmdCopyBufferToImage`. Image data can be blitted (with or without scaling and filtering) with `vkCmdBlitImage`. Multisampled images can be resolved to a non-multisampled image with `vkCmdResolveImage`.

18.1. 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 `VK_IMAGE_LAYOUT_GENERAL` or `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` layout. Destination image subresources must be in the `VK_IMAGE_LAYOUT_GENERAL` or `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 `VK_IMAGE_LAYOUT_GENERAL` layout.
- Source buffers must have been created with the `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage bit enabled and destination buffers must have been created with the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage bit enabled.
- Source images must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` set in `VkImageCreateInfo::usage`.
- Destination images must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` set in `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.
18.2. Copying Data Between Buffers

To copy data between buffer objects, call:

```c
void vkCmdCopyBuffer(
    VkCommandBuffer commandBuffer,  
    VkBuffer srcBuffer,  
    VkBuffer dstBuffer,  
    uint32_t regionCount,  
    const VkBufferCopy* pRegions);
```

- `commandBuffer` is the command buffer into which the command will be recorded.
- `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 `VkBufferCopy` structures specifying the regions to copy.

Each region in `pRegions` is copied from the source buffer to the same region of the destination buffer. `srcBuffer` and `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

- The `srcOffset` member of each element of `pRegions` must be less than the size of `srcBuffer`.
- The `dstOffset` member of each element of `pRegions` must be less than the size of `dstBuffer`.
- The `size` member of each element of `pRegions` must be less than or equal to the size of `srcBuffer` minus `srcOffset`.
- The `size` member of each element of `pRegions` must be less than or equal to the size of `dstBuffer` minus `dstOffset`.
- The union of the source regions, and the union of the destination regions, specified by the elements of `pRegions`, must not overlap in memory.
- `srcBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` usage flag.
- If `srcBuffer` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.
- `dstBuffer` must have been created with `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.
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `srcBuffer` must be a valid `VkBuffer` handle
- `dstBuffer` must be a valid `VkBuffer` handle
- `pRegions` must be a valid pointer to an array of `regionCount` valid `VkBufferCopy` structures
- `commandBuffer` must be in the recording state

- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- This command must only be called outside of a render pass instance
- `regionCount` must be greater than 0
- 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:

```c
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.
18.3. 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:

```c
void vkCmdCopyImage(
    VkCommandBuffer commandBuffer,
    VkImage srcImage,
    VkImageLayout srcImageLayout,
    VkImage dstImage,
    VkImageLayout dstImageLayout,
    uint32_t regionCount,
    const VkImageCopy* 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.

`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 $4 \times 4$ 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 $4 \times 4$ texels (using 4 bits per texel).

When copying between compressed and uncompressed formats the \textit{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 \textit{extent} must be a multiple of the compressed texel block dimension. There is one exception to this rule which is \textbf{required} to handle compressed images created with dimensions that are not a multiple of the compressed texel block dimensions: if the \textit{srcImage} is compressed, then:

- If \textit{extent.width} is not a multiple of the compressed texel block width, then $(\textit{extent.width} + \textit{srcOffset.x})$ must equal the image subresource width.
- If \textit{extent.height} is not a multiple of the compressed texel block height, then $(\textit{extent.height} + \textit{srcOffset.y})$ must equal the image subresource height.
- If \textit{extent.depth} is not a multiple of the compressed texel block depth, then $(\textit{extent.depth} + \textit{srcOffset.z})$ must equal the image subresource depth.

Similarly, if the \textit{dstImage} is compressed, then:

- If \textit{extent.width} is not a multiple of the compressed texel block width, then $(\textit{extent.width} + \textit{dstOffset.x})$ must equal the image subresource width.
- If \textit{extent.height} is not a multiple of the compressed texel block height, then $(\textit{extent.height} + \textit{dstOffset.y})$ must equal the image subresource height.
- If \textit{extent.depth} is not a multiple of the compressed texel block depth, then $(\textit{extent.depth} + \textit{dstOffset.z})$ 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.

\textbf{vkCmdCopyImage} can be used to copy image data between multisample images, but both images must have the same number of samples.
Valid Usage

- The union of all source regions, and the union of all destination regions, specified by the elements of \texttt{pRegions}, must not overlap in memory

- \texttt{srcImage} must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_SRC_BIT} usage flag

- If \texttt{srcImage} is non-sparse then it must be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- \texttt{srcImageLayout} must specify the layout of the image subresources of \texttt{srcImage} specified in \texttt{pRegions} at the time this command is executed on a \texttt{VkDevice}

- \texttt{srcImageLayout} must be \texttt{VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_GENERAL}

- \texttt{dstImage} must have been created with \texttt{VK_IMAGE_USAGE_TRANSFER_DST_BIT} usage flag

- If \texttt{dstImage} is non-sparse then it must be bound completely and contiguously to a single \texttt{VkDeviceMemory} object

- \texttt{dstImageLayout} must specify the layout of the image subresources of \texttt{dstImage} specified in \texttt{pRegions} at the time this command is executed on a \texttt{VkDevice}

- \texttt{dstImageLayout} must be \texttt{VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL} or \texttt{VK_IMAGE_LAYOUT_GENERAL}

- The \texttt{VkFormat} of each of \texttt{srcImage} and \texttt{dstImage} must be compatible, as defined above

- The sample count of \texttt{srcImage} and \texttt{dstImage} match

- 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

- 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

- The \texttt{srcSubresource.baseArrayLayer + 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

- The \texttt{dstSubresource.baseArrayLayer + 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

- 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}

- 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}
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 `VkImageCopy` structures
- `commandBuffer` **must** be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support transfer, graphics, or compute 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>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 `VkImageCopy` structure is defined as:

```c
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*.

Copies are done layer by layer starting with **baseArrayLayer** member of **srcSubresource** for the source and **dstSubresource** for the destination. **layerCount** layers are copied to the destination image.
Valid Usage

- The aspectMask member of srcSubresource and dstSubresource must match.
- The layerCount member of srcSubresource and dstSubresource must match.
- If either of the calling command’s srcImage or dstImage parameters are of VkImageType VK_IMAGE_TYPE_3D, the baseArrayLayer and layerCount members of both srcSubresource and dstSubresource must be 0 and 1, respectively.
- The aspectMask member of srcSubresource must specify aspects present in the calling command’s srcImage.
- The aspectMask member of dstSubresource must specify aspects present in the calling command’s dstImage.
- srcOffset.x and (extent.width + srcOffset.x) must both be greater than or equal to 0 and less than or equal to the source image subresource width.
- srcOffset.y and (extent.height + srcOffset.y) must both be greater than or equal to 0 and less than or equal to the source image subresource height.
- If the calling command’s srcImage is of type VK_IMAGE_TYPE_1D, then srcOffset.y must be 0 and extent.height must be 1.
- srcOffset.z and (extent.depth + srcOffset.z) must both be greater than or equal to 0 and less than or equal to the source image subresource depth.
- If the calling command’s srcImage is of type VK_IMAGE_TYPE_1D, then srcOffset.z must be 0 and extent.depth must be 1.
- If the calling command’s dstImage is of type VK_IMAGE_TYPE_1D, then dstOffset.z must be 0 and extent.depth must be 1.
- If the calling command’s srcImage is of type VK_IMAGE_TYPE_2D, then srcOffset.z must be 0.
- If the calling command’s dstImage is of type VK_IMAGE_TYPE_2D, then dstOffset.z must be 0.
- If the calling command’s srcImage or dstImage is of type VK_IMAGE_TYPE_2D, then extent.depth must be 1.
- dstOffset.x and (extent.width + dstOffset.x) must both be greater than or equal to 0 and less than or equal to the destination image subresource width.
- dstOffset.y and (extent.height + dstOffset.y) must both be greater than or equal to 0 and less than or equal to the destination image subresource height.
- If the calling command’s dstImage is of type VK_IMAGE_TYPE_1D, then dstOffset.y must be 0 and extent.height must be 1.
- dstOffset.z and (extent.depth + dstOffset.z) must both be greater than or equal to 0 and less than or equal to the destination image subresource depth.
- If the calling command’s srcImage is a compressed image, all members of srcOffset must be a multiple of the corresponding dimensions of the compressed texel block.
- If the calling command’s srcImage is a compressed image, extent.width must be a multiple of the compressed texel block width or (extent.width + srcOffset.x) must equal the source image subresource width.
• If the calling command’s `srcImage` is a compressed image, `extent.height` must be a multiple of the compressed texel block height or `(extent.height + srcOffset.y)` must equal the source image subresource height.

• If the calling command’s `srcImage` is a compressed image, `extent.depth` must be a multiple of the compressed texel block depth or `(extent.depth + srcOffset.z)` must equal the source image subresource depth.

• If the calling command’s `dstImage` is a compressed format image, all members of `dstOffset` must be a multiple of the corresponding dimensions of the compressed texel block.

• If the calling command’s `dstImage` is a compressed format image, `extent.width` must be a multiple of the compressed texel block width or `(extent.width + dstOffset.x)` must equal the destination image subresource width.

• If the calling command’s `dstImage` is a compressed format image, `extent.height` must be a multiple of the compressed texel block height or `(extent.height + dstOffset.y)` must equal the destination image subresource height.

• If the calling command’s `dstImage` is a compressed format image, `extent.depth` must be a multiple of the compressed texel block depth or `(extent.depth + dstOffset.z)` must equal the destination image subresource depth.

**Valid Usage (Implicit)**

• `srcSubresource` must be a valid `VkImageSubresourceLayers` structure.

• `dstSubresource` must be a valid `VkImageSubresourceLayers` structure.

The `VkImageSubresourceLayers` structure is defined as:

```c
typedef struct VkImageSubresourceLayers {
    VkImageAspectFlags    aspectMask;
    uint32_t              mipmapLevel;
    uint32_t              baseArrayLayer;
    uint32_t              layerCount;
} VkImageSubresourceLayers;
```

• `aspectMask` is a combination of `VkImageAspectFlagBits`, selecting the color, depth and/or stencil aspects to be copied.

• `mipmapLevel` is the mipmap level to copy from.

• `baseArrayLayer` and `layerCount` are the starting layer and number of layers to copy.
Valid Usage

- 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`.
- `aspectMask` **must** not contain `VK_IMAGE_ASPECT_METADATA_BIT`.
- `layerCount` **must** be greater than 0.

Valid Usage (Implicit)

- `aspectMask` **must** be a valid combination of `VkImageAspectFlagBits` values.
- `aspectMask` **must** not be 0.

18.4. Copying Data Between Buffers and Images

To copy data from a buffer object to an image object, call:

```c
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.
Valid Usage

- **srcBuffer** must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of **pRegions**
- The image region specified by each element of **pRegions** must be a region that is contained within **dstImage**
- The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, must not overlap in memory
- **srcBuffer** must have been created with **VK_BUFFER_USAGE_TRANSFER_SRC_BIT** usage flag
- If **srcBuffer** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object
- **dstImage** must have been created with **VK_IMAGE_USAGE_TRANSFER_DST_BIT** usage flag
- If **dstImage** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object
- **dstImage** must have a sample count equal to **VK_SAMPLE_COUNT_1_BIT**
- **dstImageLayout** must specify the layout of the image subresources of **dstImage** specified in **pRegions** at the time this command is executed on a **VkDevice**
- **dstImageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**
- 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**
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `srcBuffer` must be a valid `VkBuffer` handle
- `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 `VkBufferImageCopy` structures
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- This command must only be called outside of a render pass instance
- `regionCount` must be greater than 0
- Each of `commandBuffer`, `dstImage`, 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, Compute</td>
<td></td>
</tr>
</tbody>
</table>

To copy data from an image object to a buffer object, call:

```c
void vkCmdCopyImageToBuffer(
    VkCommandBuffer commandBuffer,  
    VkImage srcImage,              
    VkImageLayout srcImageLayout,  
    VkBuffer dstBuffer,            
    uint32_t regionCount,          
    const VkBufferImageCopy* pRegions);
```

Chapter 18. Copy Commands | 435
• **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.

---

**Valid Usage**

- The image region specified by each element of **pRegions** must be a region that is contained within **srcImage**
- **dstBuffer** must be large enough to contain all buffer locations that are accessed according to Buffer and Image Addressing, for each element of **pRegions**
- The union of all source regions, and the union of all destination regions, specified by the elements of **pRegions**, must not overlap in memory
- **srcImage** must have been created with **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** usage flag
- If **srcImage** is non-sparse then it must be bound completely and contiguously to a single **VkDeviceMemory** object
- **srcImage** must have a sample count equal to **VK_SAMPLE_COUNT_1_BIT**
- **srcImageLayout** must specify the layout of the image subresources of **srcImage** specified in **pRegions** at the time this command is executed on a **VkDevice**
- **srcImageLayout** must be **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL** or **VK_IMAGE_LAYOUT_GENERAL**
- **dstBuffer** must have been created with **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
- 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**
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `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 `VkBufferImageCopy` structures
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support transfer, graphics, or compute operations
- This command must only be called outside of a render pass instance
- `regionCount` must be greater than 0
- 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:
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 **VK_FORMAT_X8_D24_UNORM_PACK32** or **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.

---

**Note**

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.

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, 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.
Valid Usage

- If the calling command's `VkImage` parameter's format is not a depth/stencil format, then `bufferOffset` must be a multiple of the format's texel block size
- `bufferLength` must be a multiple of 4
- `bufferRowLength` must be 0, or greater than or equal to the `width` member of `imageExtent`
- `bufferImageHeight` must be 0, or greater than or equal to the `height` member of `imageExtent`
- `imageOffset.x` and `(imageExtent.width + imageOffset.x)` must both be greater than or equal to 0 and less than or equal to the image subresource width
- `imageOffset.y` and `(imageExtent.height + imageOffset.y)` must both be greater than or equal to 0 and less than or equal to the image subresource height
- If the calling command's `srcImage` (vkCmdCopyImageToBuffer) or `dstImage` (vkCmdCopyBufferToImage) is of type `VK_IMAGE_TYPE_1D`, then `imageOffset.y` must be 0 and `imageExtent.height` must be 1
- `imageOffset.z` and `(imageExtent.depth + imageOffset.z)` must both be greater than or equal to 0 and less than or equal to the image subresource depth
- If the calling command's `srcImage` (vkCmdCopyImageToBuffer) or `dstImage` (vkCmdCopyBufferToImage) is of type `VK_IMAGE_TYPE_1D` or `VK_IMAGE_TYPE_2D`, then `imageOffset.z` must be 0 and `imageExtent.depth` must be 1
- If the calling command's `VkImage` parameter is a compressed image, `bufferRowLength` must be a multiple of the compressed texel block width
- If the calling command's `VkImage` parameter is a compressed image, `bufferImageHeight` must be a multiple of the compressed texel block height
- If the calling command's `VkImage` parameter is a compressed image, all members of `imageOffset` must be a multiple of the corresponding dimensions of the compressed texel block
- If the calling command's `VkImage` parameter is a compressed image, `bufferOffset` must be a multiple of the compressed texel block size in bytes
- If the calling command's `VkImage` parameter is a compressed image, `imageExtent.width` must be a multiple of the compressed texel block width or `(imageExtent.width + imageOffset.x)` must equal the image subresource width
- If the calling command's `VkImage` parameter is a compressed image, `imageExtent.height` must be a multiple of the compressed texel block height or `(imageExtent.height + imageOffset.y)` must equal the image subresource height
- If the calling command's `VkImage` parameter is a compressed image, `imageExtent.depth` must be a multiple of the compressed texel block depth or `(imageExtent.depth + imageOffset.z)` must equal the image subresource depth
- The `aspectMask` member of `imageSubresource` must specify aspects present in the calling command's `VkImage` parameter
- The `aspectMask` member of `imageSubresource` must only have a single bit set
• If the calling command's `VkImage` parameter is of `VkImageType VK_IMAGE_TYPE_3D`, the `baseArrayLayer` and `layerCount` members of `imageSubresource` must be 0 and 1, respectively.

• When copying to the depth aspect of an image subresource, the data in the source buffer must be in the range [0,1].

Valid Usage (Implicit)

• `imageSubresource` must be a valid `VkImageSubresourceLayers` structure.

18.4.1. 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 /= compressedTexelBlockWidth;
imageHeight /= 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.

### 18.5. 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:

  \[
  u_{\text{offset}} = u_{\text{base}} - x_{\text{dst0}}
  \]

  \[
  v_{\text{offset}} = v_{\text{base}} - y_{\text{dst0}}
  \]
\[ w_{\text{offset}} = w_{\text{base}} - z_{\text{dst}0} \]

\[ a_{\text{offset}} = a - \text{baseArrayCount}_{\text{dst}} \]

- The scale is determined from the source and destination regions, and applied to the offset coordinates:

\[ \text{scale}_u = \frac{x_{\text{src}1} - x_{\text{src}0}}{x_{\text{dst}1} - x_{\text{dst}0}} \]

\[ \text{scale}_v = \frac{y_{\text{src}1} - y_{\text{src}0}}{y_{\text{dst}1} - y_{\text{dst}0}} \]

\[ \text{scale}_w = \frac{z_{\text{src}1} - z_{\text{src}0}}{z_{\text{dst}1} - z_{\text{dst}0}} \]

\[ u_{\text{scaled}} = u_{\text{offset}} \times \text{scale}_u \]

\[ v_{\text{scaled}} = v_{\text{offset}} \times \text{scale}_v \]

\[ w_{\text{scaled}} = w_{\text{offset}} \times \text{scale}_w \]

- Finally the source offset is added to the scaled coordinates, to determine the final unnormalized coordinates used to sample from \textit{srcImage}:

\[ u = u_{\text{scaled}} + x_{\text{src}0} \]

\[ v = v_{\text{scaled}} + y_{\text{src}0} \]

\[ w = w_{\text{scaled}} + z_{\text{src}0} \]

\[ q = \text{mipLevel} \]

\[ a = a_{\text{offset}} + \text{baseArrayCount}_{\text{src}} \]

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 \textit{VK_SAMPLER_MIPMAP_MODE_NEAREST} and an address mode of \textit{VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE}. 

444 | Chapter 18. 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.

3D textures are blitted slice by slice. Slices in the source region bounded by `srcOffsets[0].z` and `srcOffsets[1].z` are copied to slices in the destination region bounded by `dstOffsets[0].z` and `dstOffsets[1].z`. For each destination slice, a source $z$ coordinate is linearly interpolated between `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. If `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

- The source region specified by each element of `pRegions` **must** be a region that is contained within `srcImage`.
- The destination region specified by each element of `pRegions` **must** be a region that is contained within `dstImage`.
- 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.
- The format features of `srcImage` **must** contain `VK_FORMAT_FEATURE_BLIT_SRC_BIT`.
- `srcImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag.
- If `srcImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.
- `srcImageLayout` **must** specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.
- `srcImageLayout` **must** be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.
- The format features of `dstImage` **must** contain `VK_FORMAT_FEATURE_BLIT_DST_BIT`.
- `dstImage` **must** have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag.
- If `dstImage` is non-sparse then it **must** be bound completely and contiguously to a single `VkDeviceMemory` object.
- `dstImageLayout` **must** specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.
- `dstImageLayout` **must** be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.
- 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`.
- 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`.
- If either of `srcImage` or `dstImage` was created with a depth/stencil format, the other **must** have exactly the same format.
- If `srcImage` was created with a depth/stencil format, `filter` **must** be `VK_FILTER_NEAREST`.
- `srcImage` **must** have been created with a `samples` value of `VK_SAMPLE_COUNT_1_BIT`.
- `dstImage` **must** have been created with a `samples` value of `VK_SAMPLE_COUNT_1_BIT`.
- If `filter` is `VK_FILTER_LINEAR`, then the format features of `srcImage` **must** contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.
- The `srcSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.
- The `dstSubresource.mipLevel` member of each element of `pRegions` **must** be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.
- The `srcSubresource.baseArrayLayer + srcSubresource.layerCount` of each element of `pRegions` **must** be less than or equal to the `arrayLayers` specified in `VkImageCreateInfo`.


when \texttt{srcImage} was created

- 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

\begin{center}
\textbf{Valid Usage (Implicit)}
\end{center}

- \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle
- \texttt{srcImage} must be a valid \texttt{VkImage} handle
- \texttt{srcImageLayout} must be a valid \texttt{VkImageLayout} value
- \texttt{dstImage} must be a valid \texttt{VkImage} handle
- \texttt{dstImageLayout} must be a valid \texttt{VkImageLayout} value
- \texttt{pRegions} must be a valid pointer to an array of \texttt{regionCount} valid \texttt{VkImageBlit} structures
- \texttt{filter} must be a valid \texttt{VkFilter} value
- \texttt{commandBuffer} must be in the recording state
- The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- \texttt{regionCount} must be greater than 0
- Each of \texttt{commandBuffer}, \texttt{dstImage}, and \texttt{srcImage} must have been created, allocated, or retrieved from the same \texttt{VkDevice}

\begin{center}
\textbf{Host Synchronization}
\end{center}

- 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

\begin{center}
\textbf{Command Properties}
\end{center}

<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 \texttt{VkImageBlit} structure is defined as:
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 the specified source and destination regions.
Valid Usage

- The aspectMask member of srcSubresource and dstSubresource must match.
- The layerCount member of srcSubresource and dstSubresource must match.
- If either of the calling command’s srcImage or dstImage parameters are of VkImageType VK_IMAGE_TYPE_3D, the baseArrayLayer and layerCount members of both srcSubresource and dstSubresource must be 0 and 1, respectively.
- The aspectMask member of srcSubresource must specify aspects present in the calling command’s srcImage.
- The aspectMask member of dstSubresource must specify aspects present in the calling command’s dstImage.
- srcOffset[0].x and srcOffset[1].x must both be greater than or equal to 0 and less than or equal to the source image subresource width.
- srcOffset[0].y and srcOffset[1].y must both be greater than or equal to 0 and less than or equal to the source image subresource height.
- If the calling command’s srcImage is of type VK_IMAGE_TYPE_1D, then srcOffset[0].y must be 0 and srcOffset[1].y must be 1.
- srcOffset[0].z and srcOffset[1].z must both be greater than or equal to 0 and less than or equal to the source image subresource depth.
- If the calling command’s srcImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then srcOffset[0].z must be 0 and srcOffset[1].z must be 1.
- dstOffset[0].x and dstOffset[1].x must both be greater than or equal to 0 and less than or equal to the destination image subresource width.
- dstOffset[0].y and dstOffset[1].y must both be greater than or equal to 0 and less than or equal to the destination image subresource height.
- If the calling command’s dstImage is of type VK_IMAGE_TYPE_1D, then dstOffset[0].y must be 0 and dstOffset[1].y must be 1.
- dstOffset[0].z and dstOffset[1].z must both be greater than or equal to 0 and less than or equal to the destination image subresource depth.
- If the calling command’s dstImage is of type VK_IMAGE_TYPE_1D or VK_IMAGE_TYPE_2D, then dstOffset[0].z must be 0 and dstOffset[1].z must be 1.

Valid Usage (Implicit)

- srcSubresource must be a valid VkImageSubresourceLayers structure.
- dstSubresource must be a valid VkImageSubresourceLayers structure.
18.6. Resolving Multisample Images

To resolve a multisample image to a non-multisample image, call:

```c
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`.

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

- The source region specified by each element of `pRegions` must be a region that is contained within `srcImage`.
- The destination region specified by each element of `pRegions` must be a region that is contained within `dstImage`.
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not overlap in memory.
- If `srcImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.
- `srcImage` must have a sample count equal to any valid sample count value other than `VK_SAMPLE_COUNT_1_BIT`.
- If `dstImage` is non-sparse then it must be bound completely and contiguously to a single `VkDeviceMemory` object.
- `dstImage` must have a sample count equal to `VK_SAMPLE_COUNT_1_BIT`.
- `srcImageLayout` must specify the layout of the image subresources of `srcImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.
- `srcImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.
- `dstImageLayout` must specify the layout of the image subresources of `dstImage` specified in `pRegions` at the time this command is executed on a `VkDevice`.
- `dstImageLayout` must be `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` or `VK_IMAGE_LAYOUT_GENERAL`.
- The format features of `dstImage` must contain `VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT`.
- `srcImage` and `dstImage` must have been created with the same image format.
- The `srcSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `srcImage` was created.
- The `dstSubresource.mipLevel` member of each element of `pRegions` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `dstImage` was created.
- 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 `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.
Valid Usage (Implicit)

- \texttt{commandBuffer} must be a valid \texttt{VkCommandBuffer} handle
- \texttt{srcImage} must be a valid \texttt{VkImage} handle
- \texttt{srcImageLayout} must be a valid \texttt{VkImageLayout} value
- \texttt{dstImage} must be a valid \texttt{VkImage} handle
- \texttt{dstImageLayout} must be a valid \texttt{VkImageLayout} value
- \texttt{pRegions} must be a valid pointer to an array of \texttt{regionCount} valid \texttt{VkImageResolve} structures
- \texttt{commandBuffer} must be in the recording state
- The \texttt{VkCommandPool} that \texttt{commandBuffer} was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- \texttt{regionCount} must be greater than 0
- Each of \texttt{commandBuffer}, \texttt{dstImage}, and \texttt{srcImage} 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>Transfer</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The \texttt{VkImageResolve} structure is defined as:

```c
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**

- The **aspectMask** member of **srcSubresource** and **dstSubresource** must only contain **VK_IMAGE_ASPECT_COLOR_BIT**

- The **layerCount** member of **srcSubresource** and **dstSubresource** must match

- If either of the calling command's **srcImage** or **dstImage** parameters are of **VkImageType** **VK_IMAGE_TYPE_3D**, the **baseArrayLayer** and **layerCount** members of both **srcSubresource** and **dstSubresource** must be 0 and 1, respectively

- **srcOffset.x** and (**extent.width + srcOffset.x**) must both be greater than or equal to 0 and less than or equal to the source image subresource width

- **srcOffset.y** and (**extent.height + srcOffset.y**) must both be greater than or equal to 0 and less than or equal to the source image subresource height

- If the calling command's **srcImage** is of type **VK_IMAGE_TYPE_1D**, then **srcOffset.y** must be 0 and **extent.height** must be 1

- **srcOffset.z** and (**extent.depth + srcOffset.z**) must both be greater than or equal to 0 and less than or equal to the source image subresource depth

- If the calling command's **srcImage** is of type **VK_IMAGE_TYPE_1D** or **VK_IMAGE_TYPE_2D**, then **srcOffset.z** must be 0 and **extent.depth** must be 1

- **dstOffset.x** and (**extent.width + dstOffset.x**) must both be greater than or equal to 0 and less than or equal to the destination image subresource width

- **dstOffset.y** and (**extent.height + dstOffset.y**) must both be greater than or equal to 0 and less than or equal to the destination image subresource height

- If the calling command's **dstImage** is of type **VK_IMAGE_TYPE_1D**, then **dstOffset.y** must be 0 and **extent.height** must be 1

- **dstOffset.z** and (**extent.depth + dstOffset.z**) must both be greater than or equal to 0 and less than or equal to the destination image subresource depth

- If the calling command's **dstImage** is of type **VK_IMAGE_TYPE_1D** or **VK_IMAGE_TYPE_2D**, then **dstOffset.z** must be 0 and **extent.depth** must be 1

---

**Valid Usage (Implicit)**

- **srcSubresource** must be a valid **VkImageSubresourceLayers** structure

- **dstSubresource** must be a valid **VkImageSubresourceLayers** structure
Chapter 19. 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.

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
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 an extension-specific 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`, 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

• 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`.

• 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`.

• If the tessellation shaders feature is not enabled, topology must not be `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`.

Valid Usage (Implicit)

• `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO`.

• `pNext` must be `NULL`.

• `flags` must be `0`.

• `topology` must be a valid `VkPrimitiveTopology` value.

```c
typedef VkFlags VkPipelineInputAssemblyStateCreateFlags;
```

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

19.1. 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.

The primitive topologies defined by `VkPrimitiveTopology` are:
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.
• 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 Number</th>
<th>Provoking Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Sequence position of a vertex within the provided vertex data.</td>
<td>Provoking vertex within the main primitive. The arrow points along an edge of the relevant primitive, following winding order. Used in flat shading.</td>
</tr>
<tr>
<td>Primitive Edge</td>
<td>An edge connecting the points of a main primitive.</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></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.

19.1.1. 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 `vertexCount`.

19.1.2. 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 provoking vertex for \( p_i \) is \( v_{2i} \). The number of primitives generated is equal to \( \lfloor \text{vertexCount}/2 \rfloor \).
19.1.3. 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 provoking vertex for \( p_i \) is \( v_i \). The number of primitives generated is equal to \( \max(0, \text{vertexCount} - 1) \).

19.1.4. 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 provoking vertex for \( p_i \) is \( v_{3i} \). The number of primitives generated is equal to \( \left\lfloor \frac{\text{vertexCount}}{3} \right\rfloor \).

19.1.5. 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 provoking vertex for \( p_i \) is \( v_i \). The number of primitives generated is equal to \( \max(0, \text{vertexCount} - 2) \).

Note

The ordering of the vertices in each successive triangle is reversed, so that the winding order is consistent throughout the strip.
19.1.6. 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 provoking vertex for \( p_i \) is \( v_{i+1} \). The number of primitives generated is equal to \( \max(0, \text{vertexCount}-2) \).

![Diagram of Triangle Fans](image)

19.1.7. 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 provoking vertex for \( p_i \) is \( v_{4i+1} \). The number of primitives generated is equal to \( \lfloor \text{vertexCount}/4 \rfloor \).

![Diagram of Line Lists With Adjacency](image)

19.1.8. 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 provoking vertex for \( p_i \) is \( v_{i+1} \). The number of primitives generated is equal to \( \max(0, \text{vertexCount}-3) \).
19.1.9. 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 provoking vertex for \( p_i \) is \( v_{6i} \). The number of primitives generated is equal to \( \lfloor \frac{\text{vertexCount}}{6} \rfloor \).

19.1.10. 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:

\[ p_i = \{v_{2i}, v_{2i+1}, v_{2i+2}, v_{2i+6}, v_{2i+4}, v_{2i+3}\} \text{ when } i=0 \]

\[ p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+6}, v_{2i+2}, v_{2i+5}\} \text{ when } i>0, i<n-1, \text{ and } i\%2=1 \]

\[ p_i = \{v_{2i}, v_{2i+2}, v_{2i+3}, v_{2i+6}, v_{2i+4}, v_{2i+5}\} \text{ when } i>0, i<n-1, \text{ and } i\%2=0 \]

\[ p_i = \{v_{2i}, v_{2i+3}, v_{2i+4}, v_{2i+5}, v_{2i+2}, v_{2i+1}\} \text{ when } i=n-1 \text{ and } i\%2=1 \]
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.

\begin{itemize}
  \item Note
    \begin{itemize}
      \item The ordering of the vertices in each successive triangle is altered so that the winding order is consistent throughout the strip.
    \end{itemize}
\end{itemize}

The provoking vertex for p_i is always v_{2i}.

19.1.11. 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 \( \lfloor \frac{\text{vertexCount}}{m} \rfloor \).
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.

19.2. 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.

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, 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.

19.3. 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`

• 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`

To bind an index buffer to a command buffer, call:

```c
void vkCmdBindIndexBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset,
    VkIndexType                                 indexType);
```

• `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

• `offset` must be less than the size of `buffer`

• 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`

• `buffer` must have been created with the `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` flag

• If `buffer` is non-sparse then it must be bound completely and contiguous to a single `VkDeviceMemory` object
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `buffer` must be a valid `VkBuffer` handle
- `indexType` must be a valid `VkIndexType` value
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- 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>Graphics</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
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0,
    VK_INDEX_TYPE_UINT32 = 1,
} 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.

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
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

- If a VkImageView is sampled with VK_FILTER_LINEAR as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT.

- 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.

- 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.

- 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 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.

• 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.

• 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.

• Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set.

• 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.

• 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.

• 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.

• 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)

• commandBuffer must be a valid VkCommandBuffer handle.

• commandBuffer must be in the recording state.

• The VkCommandPool that commandBuffer was allocated from must support graphics operations.

• 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.
To record an indexed draw, call:

```c
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_UINT16`) and have `vertexOffset` added to them, before being supplied as the `vertexIndex` value.

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

- If a `VkImageView` is sampled with `VK_FILTER_LINEAR` as a result of this command, then the image view's format features must contain `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

- 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`.

- 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.

- 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 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.

• 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.

• 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.

• Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set.

• 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.

• 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.

• 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.

• 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.

• (indexSize * (firstIndex + indexCount) + 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)**

- commandBuffer must be a valid VkCommandBuffer handle.
- commandBuffer must be in the recording state.
- The VkCommandPool that commandBuffer was allocated from must support graphics operations.
- 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.
To record a non-indexed indirect draw, call:

```c
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

- If a \( \text{VkImageView} \) is sampled with \( \text{VK_FILTER_LINEAR} \) as a result of this command, then the image view's format features must contain \( \text{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT} \).

- If a \( \text{VkImageView} \) is accessed using atomic operations as a result of this command, then the image view's format features must contain \( \text{VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT} \).

- For each set \( n \) that is statically used by the \( \text{VkPipeline} \) 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 \( \text{VkPipelineLayout} \) that is compatible for set \( n \), with the \( \text{VkPipelineLayout} \) used to create the current \( \text{VkPipeline} \), as described in Pipeline Layout Compatibility.

- For each push constant that is statically used by the \( \text{VkPipeline} \) 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 \( \text{VkPipelineLayout} \) that is compatible for push constants, with the \( \text{VkPipelineLayout} \) used to create the current \( \text{VkPipeline} \), as described in Pipeline Layout Compatibility.

- Descriptors in each bound descriptor set, specified via \( \text{vkCmdBindDescriptorSets} \), must be valid if they are statically used by the \( \text{VkPipeline} \) 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 \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command requires any dynamic state, that state must have been set for \( \text{commandBuffer} \), 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 \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command, since that pipeline was bound.

- If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command accesses a \( \text{VkSampler} \) object that uses unnormalized coordinates, that sampler must not be used to sample from any \( \text{VkImage} \) with a \( \text{VkImageView} \) of the type \( \text{VK_IMAGE_VIEW_TYPE_3D}, \text{VK_IMAGE_VIEW_TYPE_CUBE}, \text{VK_IMAGE_VIEW_TYPE_1D_ARRAY}, \text{VK_IMAGE_VIEW_TYPE_2D_ARRAY} \) or \( \text{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY} \), in any shader stage.

- If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command accesses a \( \text{VkSampler} \) object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V \( \text{OpImageSample*} \) or \( \text{OpImageSparseSample*} \) instructions with ImplicitLod, Dref or Proj in their name, in any shader stage.

- If the \( \text{VkPipeline} \) object bound to the pipeline bind point used by this command accesses a \( \text{VkSampler} \) object that uses unnormalized coordinates, that sampler must not be used with any of the SPIR-V \( \text{OpImageSample*} \) or \( \text{OpImageSparseSample*} \) 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 \( \text{VkPipeline} \) 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.

• 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.

• 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.

• Every input attachment used by the current subpass must be bound to the pipeline via a descriptor set.

• 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.

• 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.

• 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.

• 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.

• If buffer is non-sparse then it must be bound completely and contiguously to a single VkDeviceMemory object.

• buffer must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set.

• offset must be a multiple of 4.

• If the multi-draw indirect feature is not enabled, drawCount must be 0 or 1.

• drawCount must be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount.

• If the drawIndirectFirstInstance feature is not enabled, all the firstInstance members of the VkDrawIndirectCommand structures accessed by this command must be 0.

• If drawCount is greater than 1, stride must be a multiple of 4 and must be greater than or equal to sizeof(VkDrawIndirectCommand).

• If drawCount is equal to 1, ((offset + sizeof(VkDrawIndirectCommand)) must be less than or equal to the size of buffer.

• If drawCount is greater than 1, (stride × (drawCount - 1) + offset + sizeof (VkDrawIndirectCommand)) must be less than or equal to the size of buffer.
Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `buffer` must be a valid `VkBuffer` handle
- `commandBuffer` must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- 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
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

• 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.

• If the drawIndirectFirstInstance feature is not enabled, firstInstance must be 0.

To record an indexed indirect draw, call:

```c
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

- If a VkImageView is sampled with VK_FILTER_LINEAR as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT.
- 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.
- 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.
- 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 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.

• 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.

• 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.

• Every input attachment used by the current subpass **must** be bound to the pipeline via a descriptor set.

• 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.

• 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.

• 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.

• 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.

• If buffer is non-sparse then it **must** be bound completely and contiguously to a single VkDeviceMemory object.

• buffer **must** have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set.

• offset **must** be a multiple of 4.

• If the multi-draw indirect feature is not enabled, drawCount **must** be 0 or 1.

• drawCount **must** be less than or equal to VkPhysicalDeviceLimits::maxDrawIndirectCount.

• If drawCount is greater than 1, stride **must** be a multiple of 4 and **must** be greater than or equal to sizeof(VkDrawIndexedIndirectCommand).

• If the drawIndirectFirstInstance feature is not enabled, all the firstInstance members of the VkDrawIndexedIndirectCommand structures accessed by this command **must** be 0.

• If drawCount is equal to 1, \((\text{offset} + \text{sizeof(VkDrawIndexedIndirectCommand)})\) **must** be less than or equal to the size of buffer.

• If drawCount is greater than 1, \((\text{stride} \times (\text{drawCount} - 1) + \text{offset} + \text{sizeof(VkDrawIndexedIndirectCommand)})\) **must** be less than or equal to the size of buffer.
Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `buffer` **must** be a valid `VkBuffer` handle
- `commandBuffer` **must** be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- This command **must** only be called inside of a render pass instance
- 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, Secondary</td>
<td>Inside</td>
<td>Graphics</td>
<td>Graphics</td>
</tr>
</tbody>
</table>

The `VkDrawIndexedIndirectCommand` structure is defined as:

```c
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**

- 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.

- `(indexSize * (firstIndex + indexCount) + 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`.

- If the `drawIndirectFirstInstance` feature is not enabled, `firstInstance` **must** be 0.
Chapter 20. Fixed-Function Vertex Processing

Vertex fetching is controlled via configurable state, as a logically distinct graphics pipeline stage.

20.1. 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 vkCmdBind Vertex Buffers 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.
20.1.1. 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 20. **Input attribute components accessed by 32-bit input variables**

---

SPIR-V example

```spire
...
%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
...
```
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 \texttt{VkVertexInputAttributeDescription::location}. Such matrices are treated as an array of column vectors with values taken from the input attributes identified in \textit{Input attributes accessed by 32-bit input matrix variables}. The \texttt{VkVertexInputAttributeDescription::format} \textbf{must} be specified with a \texttt{VkFormat} that corresponds to the appropriate type of column vector. The \texttt{Component} decoration \textbf{must} not be used with matrix types.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
\textbf{Data type} & \textbf{Column vector type} & \textbf{Locations consumed} & \textbf{Components consumed} \\
\hline
mat2 & two-component vector & i, i+1 & (x, y, o, o), (x, y, o, o) \\
mat2x3 & three-component vector & i, i+1 & (x, y, z, o), (x, y, z, o) \\
mat2x4 & four-component vector & i, i+1 & (x, y, z, w), (x, y, z, w) \\
mat3x2 & two-component vector & i, i+1, i+2 & (x, y, o, o), (x, y, o, o), (x, y, o, o) \\
mat3 & three-component vector & i, i+1, i+2 & (x, y, z, o), (x, y, z, o), (x, y, z, o) \\
mat3x4 & four-component vector & i, i+1, i+2 & (x, y, z, w), (x, y, z, w), (x, y, z, w) \\
mat4x2 & two-component vector & i, i+1, i+2, i+3 & (x, y, o, o), (x, y, o, o), (x, y, o, o), (x, y, o, o) \\
\hline
\end{tabular}
\end{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 `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 22. 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 22. 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 components 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 components 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>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, -,-)</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>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>three-component vector</td>
<td>i</td>
<td>unspecified</td>
<td>(x, y, z, w), (x, y, -,-)</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)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scalar</td>
<td>i</td>
<td>2</td>
<td>(o, o, z, w), (o, o, o)</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)</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>two-component vector</td>
<td>i</td>
<td>0 or unspecified</td>
<td>(x, y, z, w), (o, o, o)</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, z, w)</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)</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)</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.

### 20.2. 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
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 an extension-specific 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

- `vertexBindingDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`
- `vertexAttributeDescriptionCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributes`
- For every binding specified by each element of `pVertexAttributeDescriptions`, a `VkVertexInputBindingDescription` must exist in `pVertexBindingDescriptions` with the same value of binding
- All elements of `pVertexBindingDescriptions` must describe distinct binding numbers
- All elements of `pVertexAttributeDescriptions` must describe distinct attribute locations

Valid Usage (Implicit)

- `sType` must be `VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO`
- `pNext` must be `NULL`
- `flags` must be `0`
- If `vertexBindingDescriptionCount` is not `0`, `pVertexBindingDescriptions` must be a valid pointer to an array of `vertexBindingDescriptionCount` valid `VkVertexInputBindingDescription` structures
- If `vertexAttributeDescriptionCount` is not `0`, `pVertexAttributeDescriptions` must be a valid pointer to an array of `vertexAttributeDescriptionCount` valid `VkVertexInputAttributeDescription` structures

```c
typedef VkFlags VkPipelineVertexInputStateCreateFlags;
```

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

Each vertex input binding is specified by the `VkVertexInputBindingDescription` structure, defined as:

```c
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

• binding must be less than VkPhysicalDeviceLimits::maxVertexInputBindings
• stride must be less than or equal to VkPhysicalDeviceLimits::maxVertexInputBindingStride

Valid Usage (Implicit)

• 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
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
typedef struct VkVertexInputAttributeDescription {
    uint32_t    location;
    uint32_t    binding;
    VkFormat    format;
    uint32_t    offset;
} VkVertexInputAttributeDescription;
```

• location is the shader binding 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

- **location** must be less than `VkPhysicalDeviceLimits::maxVertexInputAttributes`
- **binding** must be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- **offset** must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputAttributeOffset`
- **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`

Valid Usage (Implicit)

- **format** must be a valid `VkFormat` value

To bind vertex buffers to a command buffer for use in subsequent draw commands, call:

```c
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.
Valid Usage

- **firstBinding** must be less than `VkPhysicalDeviceLimits::maxVertexInputBindings`
- The sum of `firstBinding` and `bindingCount` must be less than or equal to `VkPhysicalDeviceLimits::maxVertexInputBindings`
- All elements of `pOffsets` must be less than the size of the corresponding element in `pBuffers`
- All elements of `pBuffers` must have been created with the `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` flag
- Each element of `pBuffers` that is non-sparse must be bound completely and contiguously to a single `VkDeviceMemory` object
- If the `nullDescriptor` feature is not enabled, all elements of `pBuffers` must not be `VK_NULL_HANDLE`

Valid Usage (Implicit)

- **commandBuffer** must be a valid `VkCommandBuffer` handle
- **pBuffers** must be a valid pointer to an array of `bindingCount` valid or `VK_NULL_HANDLE` `VkBuffer` handles
- **pOffsets** must be a valid pointer to an array of `bindingCount` `VkDeviceSize` values
- **commandBuffer** must be in the recording state
- The `VkCommandPool` that `commandBuffer` was allocated from must support graphics operations
- `bindingCount` must be greater than 0
- 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
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`).

```
bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    vertexOffset = vertexIndex * bindingDesc.stride;
else
    vertexOffset = instanceIndex * bindingDesc.stride;

attribAddress = bufferBindingAddress + vertexOffset + attribDesc.offset;
```

For each attribute, raw data is extracted starting at `attribAddress` and is converted from the `VkVertexInputAttributeDescription::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.

### 20.3. Example

To create a graphics pipeline that uses the following vertex description:
The application could use the following set of structures:

```c
struct Vertex {
    float x, y, z, w;
    uint8_t u, v;
};
```

```c
const VkVertexInputBindingDescription binding = {
    0,          // binding
    sizeof(Vertex), // stride
    VK_VERTEX_INPUT_RATE_VERTEX // inputRate
};
const VkVertexInputAttributeDescription attributes[] = {
    {
        0,          // location
        binding.binding,  // binding
        VK_FORMAT_R32G32B32A32_SFLOAT, // format
        0            // offset
    },
    {
        1,          // location
        binding.binding,  // binding
        VK_FORMAT_R8G8_UNORM, // format
        4 * sizeof(float) // offset
    }
};
const VkPipelineVertexInputStateCreateInfo viInfo = {
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_CREATE_INFO, // sType
    NULL,        // pNext
    0,           // flags
    1,           // vertexBindingDescriptionCount
    &binding,    // pVertexBindingDescriptions
    2,           // vertexAttributeDescriptionCount
    &attributes[0] // pVertexAttributeDescriptions
};
```
Chapter 21. 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.

21.1. 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.

Tessellate 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.

- **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.

- **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 figure Domain parameterization for tessellation primitive modes (upper-left origin). The domain space has an upper-left origin.

![Domain parameterization diagram](image)

**Caption**

In the domain parameterization diagrams, the coordinates illustrate the value of \text{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.

### 21.2. 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.

### 21.3. 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 \(\text{maxLevel}\) is the implementation-dependent maximum tessellation level (\(\text{VkPhysicalDeviceLimits}::\text{maxTessellationGenerationLevel}\)). 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.

### 21.4. 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.
21.5. 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.

**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 \(v\).

21.6. 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 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 6. 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.

### 21.7. 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 mesh 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 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 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 on the top figure 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**.

### 21.8. 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 \(0, \frac{1}{n}, \frac{2}{n}, ..., \frac{n-1}{n}\).

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.

### 21.9. 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 `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.

### 21.10. Tessellation Pipeline State

The `pTessellationState` member of `VkGraphicsPipelineCreateInfo` is a pointer to a `VkPipelineTessellationStateCreateInfo` structure.

The `VkPipelineTessellationStateCreateInfo` structure is defined as:

```c
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 an extension-specific structure.
- `flags` is reserved for future use.
- `patchControlPoints` number of control points per patch.
Valid Usage

• **patchControlPoints** must be greater than zero and less than or equal to `VkPhysicalDeviceLimits::maxTessellationPatchSize`

Valid Usage (Implicit)

• **sType** must be `VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO`
• **pNext** must be `NULL`
• **flags** must be `0`

**typedef** `VkFlags VkPipelineTessellationStateCreateFlags;`

`VkPipelineTessellationStateCreateFlags` is a bitmask type for setting a mask, but is currently reserved for future use.
Chapter 22. 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.

22.1. 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.
22.2. 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 \texttt{OpExecutionMode} instruction with the \texttt{OutputPoints}, \texttt{OutputLineStrip} or \texttt{OutputTriangleStrip} modes, respectively. Each geometry shader \textbf{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 \texttt{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 \texttt{OpExecutionMode} instruction with the mode set to \texttt{OutputVertices} and the maximum number of vertices that will be produced by the geometry shader specified as a literal. Each geometry shader \textbf{must} specify a maximum output vertex count.

22.3. Multiple Invocations of Geometry Shaders

Geometry shaders \textbf{can} be invoked more than one time for each input primitive. This is known as \textit{geometry shader instancing} and is requested by including an \texttt{OpExecutionMode} instruction with \texttt{mode} specified as \texttt{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 \texttt{OpExecutionMode} instruction. The instance number is available to each invocation as a built-in input using \texttt{InvocationId}.

22.4. Geometry Shader Primitive Ordering

Limited guarantees are provided for the relative ordering of primitives produced by a geometry shader, as they pertain to \textit{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.
Chapter 23. Fixed-Function Vertex Post-Processing

After programmable vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

- Flat shading (see Flat Shading).
- Primitive clipping, including client-defined half-spaces (see Primitive Clipping).
- Shader output attribute clipping (see Clipping Shader Outputs).
- 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.

23.1. Flat Shading

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 provoking vertex of the primitive. Flat shading is applied to those vertex attributes that match fragment input attributes which are decorated as flat.

If neither geometry nor tessellation shading is active, the provoking vertex is determined by the primitive topology defined by VkPipelineInputAssemblyStateCreateInfo:topology used to execute the drawing command.

If geometry shading is active, the provoking vertex is determined by the primitive topology defined by the OutputPoints, OutputLineStrips, or OutputTriangleStrips execution mode.

If tessellation shading is active but geometry shading is not, the provoking vertex may be any of the vertices in each primitive.

23.2. 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
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.

Depth clamping is enabled or disabled via the `depthClampEnable` enable of the `VkPipelineRasterizationStateCreateInfo` structure. Depth clipping is disabled when `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.

If a point's vertex lies outside of the clip volume, the entire primitive **may** be discarded.

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 original vertices' coordinates are $P_1$ and $P_2$, then $t$ is given by
\[ P = t P_1 + (1-t) P_2. \]

\( t \) 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 lie entirely inside the clip volume, and it is discarded if every one of its edges lie entirely outside the clip volume. 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.

### 23.3. 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.

### 23.4. 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

\[
\begin{pmatrix}
  x_d \\
  y_d \\
  z_d
\end{pmatrix} = \begin{pmatrix}
  x_c \\
  y_c \\
  z_c \\
  w_c
\end{pmatrix}
\]

### 23.5. 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 \( \text{VkPhysicalDeviceLimits::maxViewports} \) minus one. The number of viewports used by a pipeline is controlled by the \( \text{viewportCount} \) member of the \( \text{VkPipelineViewportStateCreateInfo} \) structure used in pipeline creation.

The \( \text{VkPipelineViewportStateCreateInfo} \) structure is defined as:
typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType                       sType;
    const void*                           pNext;
    VkPipelineViewportStateCreateFlags    flags;
    uint32_t                              viewportCount;
    const VkViewport*                     pViewports;
    uint32_t                              scissorCount;
    const VkRect2D*                       pScissors;
} VkPipelineViewportStateCreateInfo;

• sType is the type of this structure.
• pNext is NULL or a pointer to an extension-specific structure.
• flags is reserved for future use.
• viewportCount is the number of viewports used by the pipeline.
• pViewports is a pointer to an array of VkViewport structures, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
• scissorCount is the number of scissors and must match the number of viewports.
• pScissors is a pointer to an array of VkRect2D structures defining the rectangular bounds of the scissors for the corresponding viewport. If the scissor state is dynamic, this member is ignored.

Valid Usage

• If the multiple viewports feature is not enabled, viewportCount must be 1
• If the multiple viewports feature is not enabled, scissorCount must be 1
• viewportCount must be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive
• scissorCount must be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive
• scissorCount and viewportCount must be identical
• The x and y members of offset member of any element of pScissors must be greater than or equal to 0
• Evaluation of (offset.x + extent.width) must not cause a signed integer addition overflow for any element of pScissors
• Evaluation of (offset.y + extent.height) must not cause a signed integer addition overflow for any element of pScissors
Valid Usage (Implicit)

- **sType** must be **VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO**
- **pNext** must be NULL
- **flags** must be 0
- **viewportCount** must be greater than 0
- **scissorCount** must be greater than 0

```c
typedef VkFlags VkPipelineViewportStateCreateFlags;
```

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

If a geometry shader is active and has an output variable decorated with **ViewportIndex**, the viewport transformation uses the viewport corresponding to the value assigned to **ViewportIndex** taken from an implementation-dependent vertex of each primitive. If **ViewportIndex** is outside the range zero to **viewportCount** minus one for a primitive, or if the geometry shader did not assign a value to **ViewportIndex** 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 no geometry shader is active, or if the geometry shader does not have an output decorated with **ViewportIndex**, 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:

```c
void vkCmdSetViewport(
    VkCommandBuffer commandBuffer,           commandBuffer,
    uint32_t firstViewport,                  firstViewport,
    uint32_t viewportCount,                  viewportCount,
    const VkViewport* pViewports);          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

- **firstViewport** must be less than VkPhysicalDeviceLimits::maxViewports
- The sum of **firstViewport** and **viewportCount** must be between 1 and VkPhysicalDeviceLimits::maxViewports, inclusive
- If the multiple viewports feature is not enabled, **firstViewport** must be 0
- If the multiple viewports feature is not enabled, **viewportCount** must be 1

### Valid Usage (Implicit)

- **commandBuffer** must be a valid VkCommandBuffer handle
- **pViewports** must be a valid pointer to an array of **viewportCount** valid VkViewport structures
- **commandBuffer** must be in the recording state
- The VkCommandPool that **commandBuffer** was allocated from must support graphics operations
- **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 Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>

Both VkPipelineViewportStateCreateInfo and vkCmdSetViewport use VkViewport to set the viewport transformation parameters.

The **VkViewport** structure is defined as:
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. It is valid for minDepth to be greater than or equal to maxDepth.

The framebuffer depth coordinate \( z_f \) 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 \( k \cdot \frac{1}{2^{m-1}} \), where \( k \in \{0, 1, ..., 2^m-1\} \), 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

\[
\begin{align*}
    o_x &= x + width / 2 \\
    o_y &= y + height / 2 \\
    o_z &= minDepth \\
    p_x &= width \\
    p_y &= height \\
    p_z &= maxDepth - minDepth.
\end{align*}
\]

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

• width **must** be greater than 0.0
• width **must** be less than or equal to VkPhysicalDeviceLimits::maxViewportDimensions[0]
• height **must** be greater than 0.0
• The absolute value of height **must** be less than or equal to VkPhysicalDeviceLimits::maxViewportDimensions[1]
• x **must** be greater than or equal to viewportBoundsRange[0]
• (x + width) **must** be less than or equal to viewportBoundsRange[1]
• y **must** be greater than or equal to viewportBoundsRange[0]
• (y + height) **must** be less than or equal to viewportBoundsRange[1]
• minDepth **must** be between 0.0 and 1.0, inclusive
• maxDepth **must** be between 0.0 and 1.0, inclusive
Chapter 24. Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point 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
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkPipelineRasterizationStateCreateFlags flags;
    VkBool32 depthClampEnable;
    VkBool32 rasterizerDiscardEnable;
    VkPolygonMode polygonMode;
    VkCullModeFlags cullMode;
    VkFrontFace frontFace;
    float lineWidth;
    float depthBiasConstantFactor;
    float depthBiasClamp;
    float depthBiasSlopeFactor;
} VkPipelineRasterizationStateCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
• \texttt{depthClampEnable} controls whether to clamp the fragment’s depth values as described in \textit{Depth Test}. Enabling depth clamp will also disable clipping primitives to the z planes of the frustum as described in \textit{Primitive Clipping}.

• \texttt{rasterizerDiscardEnable} controls whether primitives are discarded immediately before the rasterization stage.

• \texttt{polygonMode} is the triangle rendering mode. See \texttt{VkPolygonMode}.

• \texttt{cullMode} is the triangle facing direction used for primitive culling. See \texttt{VkCullModeFlagBits}.

• \texttt{frontFace} is a \texttt{VkFrontFace} value specifying the front-facing triangle orientation to be used for culling.

• \texttt{depthBiasEnable} controls whether to bias fragment depth values.

• \texttt{depthBiasConstantFactor} is a scalar factor controlling the constant depth value added to each fragment.

• \texttt{depthBiasClamp} is the maximum (or minimum) depth bias of a fragment.

• \texttt{depthBiasSlopeFactor} is a scalar factor applied to a fragment's slope in depth bias calculations.

• \texttt{lineWidth} is the width of rasterized line segments.

### Valid Usage

- If the \texttt{depth clamping} feature is not enabled, \texttt{depthClampEnable} must be \texttt{VK_FALSE}

- If the \texttt{non-solid fill modes} feature is not enabled, \texttt{polygonMode} must be \texttt{VK_POLYGON_MODE_FILL}

### Valid Usage (Implicit)

- \texttt{sType} must be \texttt{VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO}

- \texttt{pNext} must be \texttt{NULL}

- \texttt{flags} must be \texttt{0}

- \texttt{polygonMode} must be a valid \texttt{VkPolygonMode} value

- \texttt{cullMode} must be a valid combination of \texttt{VkCullModeFlagBits} values

- \texttt{frontFace} must be a valid \texttt{VkFrontFace} value

\begin{verbatim}
typedef VkFlags VkPipelineRasterizationStateCreateFlags;
\end{verbatim}

\texttt{VkPipelineRasterizationStateCreateFlags} is a bitmask type for setting a mask, but is currently reserved for future use.

The \texttt{VkPipelineMultisampleStateCreateInfo} structure is defined as:
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 an extension-specific 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 \( \lceil \frac{rasterizationSamples}{32} \rceil \) words.

If `pSampleMask` is NULL, it is treated as if the mask has all bits set to 1.

### Valid Usage

- If the sample rate shading feature is not enabled, `sampleShadingEnable` must be `VK_FALSE`
- If the alpha to one feature is not enabled, `alphaToOneEnable` must be `VK_FALSE`
- `minSampleShading` must be in the range \([0, 1]\)
Valid Usage (Implicit)

- \textbf{sType must} be \texttt{VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO}
- \textbf{pNext must} be \texttt{NULL}
- \textbf{flags must} be \texttt{0}
- \textbf{rasterizationSamples must} be a valid \texttt{VkSampleCountFlagBits} value
- If \texttt{pSampleMask} is not \texttt{NULL}, \texttt{pSampleMask must} be a valid pointer to an array of \texttt{VkSampleMask} values.

\begin{verbatim}
typedef VkFlags VkPipelineMultisampleStateCreateFlags;

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

The elements of the sample mask array are of type \texttt{VkSampleMask}, each representing 32 bits of coverage information:

\begin{verbatim}
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 \textbf{can also modify or replace their assigned depth values}.

When the \texttt{VK_AMD_mixed_attachment_samples} and \texttt{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 attachments, then \texttt{rasterizationSamples must} be the same as the sample count for those subpass attachments.

If the subpass for which this pipeline is being created does not use color or depth/stencil attachments, \texttt{rasterizationSamples must} follow the rules for a zero-attachment subpass.

24.1. Discarding Primitives Before Rasterization

Primitives are discarded before rasterization if the \texttt{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.
24.2. 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 primitive order.

24.3. 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 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 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 may be identical for each pixel in the framebuffer, or they may differ.

If the current pipeline includes a fragment shader with one or more variables in its interface decorated with Sample and Input, the data associated with those variables will be assigned independently for each sample. The values for each sample must be evaluated at the location of the sample. The data associated with any other variables not decorated with Sample and Input need not be evaluated independently for each sample.

A 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 have one set of samples. Each set of samples has a number of samples
determined by `VkPipelineMultisampleStateCreateInfo::rasterizationSamples`. Each sample in a set is assigned a unique sample index \( i \) in the range \([0, \text{rasterizationSamples})\).

Each sample in a fragment is also assigned a unique coverage index \( j \) in the range \([0, n * \text{rasterizationSamples})\), where \( n \) is the number of sets in the fragment. If the fragment contains a single set of samples, the coverage index is always equal to the sample index.

The coverage mask includes \( B \) bits packed into \( W \) words, defined as:

\[
B = n * \text{rasterizationSamples}
\]

\[
W = \lceil B / 32 \rceil
\]

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 `standardSampleLocations` member of `VkPhysicalDeviceLimits` is `VK_TRUE`, then the sample counts `VK_SAMPLE_COUNT_1_BIT`, `VK_SAMPLE_COUNT_2_BIT`, `VK_SAMPLE_COUNT_4_BIT`, `VK_SAMPLE_COUNT_8_BIT`, and `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 \( i \). `VK_SAMPLE_COUNT_32_BIT` and `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.
Table 23. Standard sample locations

<table>
<thead>
<tr>
<th>VK_SAMPLE_COUNT_1_BIT</th>
<th>VK_SAMPLE_COUNT_2_BIT</th>
<th>VK_SAMPLE_COUNT_4_BIT</th>
<th>VK_SAMPLE_COUNT_8_BIT</th>
<th>VK_SAMPLE_COUNT_16_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0.5,0.5)</td>
<td>(0.75,0.75)</td>
<td>(0.375,0.125)</td>
<td>(0.5625,0.3125)</td>
<td>(0.5625, 0.5625)</td>
</tr>
<tr>
<td>(0.25,0.25)</td>
<td>(0.875,0.375)</td>
<td>(0.875,0.6875)</td>
<td>(0.4375, 0.6875)</td>
<td>(0.4375, 0.3125)</td>
</tr>
<tr>
<td></td>
<td>(0.125,0.625)</td>
<td>(0.8125,0.5625)</td>
<td>(0.3125, 0.1875)</td>
<td>(0.3125, 0.625)</td>
</tr>
<tr>
<td></td>
<td>(0.625,0.875)</td>
<td>(0.1875,0.8125)</td>
<td>(0.0625, 0.4375)</td>
<td>(0.75, 0.4375)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.6875,0.9375)</td>
<td>(0.1875, 0.8125)</td>
<td>(0.625, 0.8125)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.9375,0.0625)</td>
<td>(0.6875, 0.6875)</td>
<td>(0.8125,0.1875)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.375, 0.875)</td>
<td>(0.875, 0.375)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.9375,0.25)</td>
<td>(0.875, 0.9375)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.0625, 0.0)</td>
<td>(0.0625, 0.0)</td>
</tr>
</tbody>
</table>

24.4. 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 max(\( \lfloor \text{minSampleShadingFactor} \times \text{totalSamples} \rfloor, 1 \)) unique associated data for each fragment, where \( \text{minSampleShadingFactor} \) is the minimum fraction of sample shading. \( \text{totalSamples} \) is the value of \( \text{VkPipelineMultisampleStateCreateInfo}::\text{rasterizationSamples} \) 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 \( \text{VkPipelineMultisampleStateCreateInfo}::\text{minSampleShading} \)
Otherwise, sample shading is considered disabled.

### 24.5. 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 \([\texttt{pointSizeRange}[0], \texttt{pointSizeRange}[1]]\). The value written to \texttt{PointSize} \textbf{must} be greater than zero.

Not all point sizes need be supported, but the size 1.0 \textbf{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 size 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional point sizes \textbf{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.

#### 24.5.1. 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 \texttt{s} and \texttt{t} point sprite texture coordinates vary from zero to one across the point horizontally left-to-right and top-to-bottom, respectively. The following formulas are used to evaluate \texttt{s} and \texttt{t}:

\[
\begin{align*}
    s &= \frac{1}{2} + \frac{(x_p - x_f)}{\text{size}} \\
    t &= \frac{1}{2} + \frac{(y_p - y_f)}{\text{size}}
\end{align*}
\]

where \texttt{size} is the point’s size; \((x_p,y_p)\) is the location at which the point sprite coordinates are evaluated - this \textbf{may} be the framebuffer coordinates of the fragment center, or the location of a sample; and \((x_f,y_f)\) is the exact, unrounded framebuffer coordinate of the vertex for the point.

### 24.6. Line Segments

Each line segment has an associated width. The line width is specified by the \texttt{VkPipelineRasterizationStateCreateInfo::lineWidth} property of the currently active pipeline, if the pipeline was not created with \texttt{VK_DYNAMIC_STATE_LINE_WIDTH} enabled.
Otherwise, the line width is set by calling `vkCmdSetLineWidth`:

```c
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
- If the wide lines feature is not enabled, `lineWidth` must be 1.0

### Valid Usage (Implicit)
- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `commandBuffer` must be in the recording state
- 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 size 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.
**24.6.1. Basic Line Segment Rasterization**

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 a fragment. When `rasterizationSamples` is `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

$$t = \frac{(p_r - p_a) \cdot (p_b - p_a)}{||p_b - p_a||^2}$$

(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 segments, 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 **NoPerspective** and **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 **NoPerspective** decoration is used, **linear interpolation** is performed in the same fashion as for depth values, as described above. When the **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.

The above description documents the preferred method of line rasterization, and must be used when the implementation advertises the `strictLines` limit in `VkPhysicalDeviceLimits` as `VK_TRUE`.

When `strictLines` is `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 Non strict lines, and each is 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.

![Figure 8. Non strict lines](image)

### 24.7. 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.

#### 24.7.1. 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:

\[
a = -\frac{1}{2} \sum_{i=0}^{n-1} x_i^i y_{i+1}^i - x_{i+1}^i y_i^i
\]

where \(x_i^i\) and \(y_i^i\) 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:
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;

- **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.

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**.

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
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:

\[
a = \frac{A(p_p b_p c_p)}{A(p_a p_b p_c)}, \quad b = \frac{A(p_p a_p c_p)}{A(p_a p_b p_c)}, \quad c = \frac{A(p_p a_p b_p)}{A(p_a p_b p_c)},
\]

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.

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 \( j \neq i \).
24.7.2. Polygon Mode

Possible values of the `VkPipelineRasterizationStateCreateInfo::polygonMode` property of the currently active pipeline, specifying the method of rasterization for polygons, are:

```
typedef enum VkPolygonMode {
    VK_POLYGON_MODE_FILL = 0,
    VK_POLYGON_MODE_LINE = 1,
    VK_POLYGON_MODE_POINT = 2,
} VkPolygonMode;
```

- `VK_POLYGON_MODE_POINT` specifies that polygon vertices are drawn as points.
- `VK_POLYGON_MODE_LINE` specifies that polygon edges are drawn as line segments.
- `VK_POLYGON_MODE_FILL` specifies that polygons are rendered using the polygon rasterization rules in this section.

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.

24.7.3. 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` command if depth bias state is dynamic.

```
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`, 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 an implementation-dependent constant that relates to the usable resolution 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

\[
m = \max\left(\left|\frac{\partial z_f}{\partial x_f}\right|, \left|\frac{\partial z_f}{\partial y_f}\right|\right).
\]

The minimum resolvable difference \( r \) is an implementation-dependent 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 \( z_f \) 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. 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

\[
r = 2^{e-a}
\]

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 \(
\text{dbclamp}(x) = \begin{cases} 
  x & \text{depthBiasClamp} = 0 \text{ or } \text{NaN} \\
  \min(x, \text{depthBiasClamp}) & \text{depthBiasClamp} > 0 \\
  \max(x, \text{depthBiasClamp}) & \text{depthBiasClamp} < 0
\end{cases}
\)

\( m \) 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. Fragment depth values are clamped even when
the depth buffer uses a floating-point representation.

### Valid Usage

- If the `depth bias clamping` feature is not enabled, `depthBiasClamp` must be `0.0`

### Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `commandBuffer` must be in the `recording state`
- 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>
Chapter 25. 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. Scissor test
2. Sample mask test
3. Fragment shading
4. Multisample coverage
5. Depth bounds test
6. Stencil test
7. Depth test
8. Sample counting
9. Coverage reduction

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 `OpKill`.

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.

25.1. 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 \([\text{VkRect2D::offset.x}, \text{VkRect2D::offset.x} + \text{VkRect2D::extent.x}]\), and \(y_f\) is in the range \([\text{VkRect2D::offset.y}, \text{VkRect2D::offset.y} + \text{VkRect2D::extent.y}]\). Samples with coordinates outside the scissor rectangle at the corresponding `ViewportIndex` will have their coverage set to 0.

The scissor rectangles can be set dynamically with the command:
```c
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

- `firstScissor` **must** be less than `VkPhysicalDeviceLimits::maxViewports`
- The sum of `firstScissor` and `scissorCount` **must** be between 1 and `VkPhysicalDeviceLimits::maxViewports`, inclusive
- If the `multiple viewports` feature is not enabled, `firstScissor` **must** be 0
- If the `multiple viewports` feature is not enabled, `scissorCount` **must** be 1
- The `x` and `y` members of `offset` member of any element of `pScissors` **must** be greater than or equal to 0
- Evaluation of `(offset.x + extent.width)` **must** not cause a signed integer addition overflow for any element of `pScissors`
- Evaluation of `(offset.y + extent.height)` **must** not cause a signed integer addition overflow for any element of `pScissors`

### Valid Usage (Implicit)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `pScissors` **must** be a valid pointer to an array of `scissorCount` `VkRect2D` structures
- `commandBuffer` **must** be in the **recording state**
- The `VkCommandPool` that `commandBuffer` was allocated from **must** support graphics operations
- `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, Secondary</td>
<td>Both</td>
<td>Graphics</td>
<td></td>
</tr>
</tbody>
</table>

25.2. 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.

25.3. Multisample Coverage

If a fragment shader is active and its entry point’s interface includes a built-in output variable decorated with `SampleMask`, the coverage mask is ANDed with the bits of the `SampleMask` built-in to generate a new coverage mask. 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 `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 `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.

Next, if 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.

25.4. Depth and Stencil Operations

Pipeline state controlling the depth bounds tests, stencil test, and depth test is specified through the members of the VkPipelineDepthStencilStateCreateInfo structure.

The VkPipelineDepthStencilStateCreateInfo structure is defined as:

```
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;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to an extension-specific structure.
- **flags** is reserved for future use.
- **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

- If the **depth bounds testing** feature is not enabled, `depthBoundsTestEnable` must be `VK_FALSE`.

### Valid Usage (Implicit)

- **sType** must be `VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO`
- **pNext** must be `NULL`
- **flags** must be `0`
- **depthCompareOp** must be a valid `VkCompareOp` value
- **front** must be a valid `VkStencilOpState` structure
- **back** must be a valid `VkStencilOpState` structure

#### typedef

```c
typedef VkFlags VkPipelineDepthStencilStateCreateFlags;
```

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

## 25.5. 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 `VkPipelineDepthStencilStateCreateInfo` structure during pipeline creation, or dynamically by the `vkCmdSetDepthBounds` command.

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 set the depth bounds range values call:

```c
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

- `minDepthBounds` must be between 0.0 and 1.0, inclusive
- `maxDepthBounds` must be between 0.0 and 1.0, inclusive

### Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `commandBuffer` must be in the recording state
- 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>

### 25.6. 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 stencil test is not enabled, as specified by `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 $c$, and stencil reference value $r$ 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 $r$ and $a$ are each independently combined with the compare mask $c$, using a logical AND operation to create masked reference and attachment values $r'$ and $a'$, $s_r$ 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. If the stencil test fails, VkPipelineDepthStencilStateCreateInfo::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. whether the stencil test passed or failed, and then whether the depth test passed or failed.

The stencil attachment value $a$ is then updated with the generated stencil value $s_g$ according to the write mask $w$ defined by VkPipelineDepthStencilStateCreateInfo::writeMask as:

$$ s_a = (s_a \& \neg s_w) | (s_g \& s_w) $$

The VkStencilOpState structure is defined as:

```c
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)

- `failOp` must be a valid `VkStencilOp` value
- `passOp` must be a valid `VkStencilOp` value
- `depthFailOp` must be a valid `VkStencilOp` value
- `compareOp` must be a valid `VkCompareOp` value

To dynamically set the stencil compare mask call:

```c
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_STENCIL_COMPARE_MASK` set in `VkPipelineDynamicStateCreateInfo::pDynamicStates`.

### Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values
- `faceMask` must not be 0
- `commandBuffer` must be in the recording state
- 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
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
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
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)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `faceMask` must be a valid combination of `VkStencilFaceFlagBits` values
- `faceMask` must not be 0
- `commandBuffer` must be in the recording state
- 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
void vkCmdSetStencilReference(
    VkCommandBuffer                         commandBuffer,  // commandBuffer is the command buffer into which the command will be recorded.
    VkStencilFaceFlags                     faceMask,        // faceMask is a bitmask of `VkStencilFaceFlagBits` specifying the set of stencil state for which to update the reference value, as described above for `vkCmdSetStencilCompareMask`.
    uint32_t                                reference);      // 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)

- `commandBuffer` **must** be a valid `VkCommandBuffer` handle
- `faceMask` **must** be a valid combination of `VkStencilFaceFlagBits` values
- `faceMask` **must** not be 0
- `commandBuffer` **must** be in the recording state
- 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>
</tbody>
</table>

Possible values of `VkStencilOpState::compareOp`, specifying the stencil comparison function, are:

```c
typedef enum VkCompareOp {
    VK_COMPARE_OP_NEVER = 0,
    VK_COMPARE_OP_LESS = 1,
    VK_COMPARE_OP_EQUAL = 2,
    VK_COMPARE_OP_LESS_OR_EQUAL = 3,
    VK_COMPARE_OP_GREATER = 4,
    VK_COMPARE_OP_NOT_EQUAL = 5,
    VK_COMPARE_OP_GREATER_OR_EQUAL = 6,
    VK_COMPARE_OP_ALWAYS = 7,
} VkCompareOp;
```

- **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 $A > B$.  

538 | Chapter 25. Fragment Operations
• **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:

```c
typedef enum VkStencilOp {
    VK_STENCIL_OP_KEEP = 0,
    VK_STENCIL_OP_ZERO = 1,
    VK_STENCIL_OP_REPLACE = 2,
    VK_STENCIL_OP_INCREMENT_AND_CLAMP = 3,
    VK_STENCIL_OP_DECURRENT_AND_CLAMP = 4,
    VK_STENCIL_OP_INVERT = 5,
    VK_STENCIL_OP_INCREMENT_AND_WRAP = 6,
    VK_STENCIL_OP_DECURRENT_AND_WRAP = 7,
} VkStencilOp;
```

• **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_DECURRENT_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_DECURRENT_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.

## 25.7. 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 depth test is not enabled, as specified by `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 `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 \texttt{VkPipelineRasterizationStateCreateInfo::depthClampEnable} is enabled, before the sample’s \(z_f\) is compared to \(z_a\), \(z_f\) is clamped to \([\min(n,f),\max(n,f)]\), where \(n\) and \(f\) are the \texttt{minDepth} and \texttt{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 \texttt{VkPipelineDepthStencilStateCreateInfo::depthWriteEnable}, and the comparison evaluated to true, the depth attachment value \(z_a\) is set to the sample’s depth value \(z_f\).

25.8. 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 \textit{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, sample mask, alpha to coverage, stencil, and depth tests.

25.9. Coverage Reduction

Coverage reduction generates a \textit{color sample mask} from the coverage mask, with one bit for each sample in the color attachment(s) for the subpass. If a bit in the color sample mask is 0, then blending and writing to the framebuffer are not performed for that sample.

If the fragment covers a single pixel with a number of samples equal to the number of samples in each color attachment, each color sample is associated with a unique rasterization sample, and the value of the coverage mask is assigned to the color sample mask.
Chapter 26. The Framebuffer

26.1. 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.

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;
    const void* pNext;
    VkPipelineColorBlendStateCreateFlags flags;
    VkBool32 logicOpEnable;
    VkLogicOp logicOp;
    uint32_t attachmentCount;
    const VkPipelineColorBlendAttachmentState* pAttachments;
    float blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

- **sType** is the type of this structure.
- **pNext** is NULL or a pointer to an extension-specific 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`. This value **must** equal the `colorAttachmentCount` for the subpass in which this pipeline is used.
• `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.

### Valid Usage

- If the independent blending feature is not enabled, all elements of `pAttachments` **must** be identical.
- If the logic operations feature is not enabled, `logicOpEnable` **must** be `VK_FALSE`.
- If `logicOpEnable` is `VK_TRUE`, `logicOp` **must** be a valid `VkLogicOp` value.

### Valid Usage (Implicit)

- `sType` **must** be `VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO`.
- `pNext` **must** be `NULL`.
- `flags` **must** be `0`.
- If `attachmentCount` is not `0`, `pAttachments` **must** be a valid pointer to an array of `attachmentCount` valid `VkPipelineColorBlendAttachmentState` structures.

```c
typedef VkFlags VkPipelineColorBlendStateCreateFlags;
```

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

The `VkPipelineColorBlendAttachmentState` structure is defined as:
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 ($S_s$, $S_g$, $S_b$).

- **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 color attachment.

- **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 use 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**

- 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

- 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

- 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

- 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
Valid Usage (Implicit)

- `srcColorBlendFactor` must be a valid `VkBlendFactor` value
- `dstColorBlendFactor` must be a valid `VkBlendFactor` value
- `colorBlendOp` must be a valid `VkBlendOp` value
- `srcAlphaBlendFactor` must be a valid `VkBlendFactor` value
- `dstAlphaBlendFactor` must be a valid `VkBlendFactor` value
- `alphaBlendOp` must be a valid `VkBlendOp` value
- `colorWriteMask` must be a valid combination of `VkColorComponentFlagBits` values

26.1.1. Blend Factors

The source and destination color and alpha blending factors are selected from the enum:

```c
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 each enum value is described in the table below:

<table>
<thead>
<tr>
<th><code>VkBlendFactor</code></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>VKBlendFactor</td>
<td>RGB Blend Factors ((S_r,S_g,S_b)) or ((D_r,D_g,D_b))</td>
<td>Alpha Blend Factor ((S_a) or (D_a))</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>-------------------------------------------------</td>
<td>---------------------------------</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_{s0},G_{s0},B_{s0}))</td>
<td>(A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR</td>
<td>((1-R_{s0},1-G_{s0},1-B_{s0}))</td>
<td>(1-A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_COLOR</td>
<td>((R_{d0},G_{d0},B_{d0}))</td>
<td>(A_{d})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR</td>
<td>((1-R_{d0},1-G_{d0},1-B_{d0}))</td>
<td>(1-A_{d})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA</td>
<td>((A_{s0},A_{s0},A_{s0}))</td>
<td>(A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA</td>
<td>((1-A_{s0},1-A_{s0},1-A_{s0}))</td>
<td>(1-A_{s0})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_DST_ALPHA</td>
<td>((A_{d0},A_{d0},A_{d0}))</td>
<td>(A_{d})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA</td>
<td>((1-A_{d0},1-A_{d0},1-A_{d0}))</td>
<td>(1-A_{d0})</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},A_{c},A_{c}))</td>
<td>(A_{c})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA</td>
<td>((1-A_{c},1-A_{c},1-A_{c}))</td>
<td>(1-A_{c})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC_ALPHA_SATURATE</td>
<td>((f,f,f); f = \min(A_{s0},1-A_{d}))</td>
<td>1</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC1_COLOR</td>
<td>((R_{s1},G_{s1},B_{s1}))</td>
<td>(A_{s1})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR</td>
<td>((1-R_{s1},1-G_{s1},1-B_{s1}))</td>
<td>(1-A_{s1})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_SRC1_ALPHA</td>
<td>((A_{s1},A_{s1},A_{s1}))</td>
<td>(A_{s1})</td>
</tr>
<tr>
<td>VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA</td>
<td>((1-A_{s1},1-A_{s1},1-A_{s1}))</td>
<td>(1-A_{s1})</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 \texttt{VK_DYNAMIC_STATE_BLEND_CONSTANTS} dynamic state enabled then the blend constant \((R_c,G_c,B_c,A_c)\) is specified via the \texttt{blendConstants} member of \texttt{VkPipelineColorBlendStateCreateInfo}.

Otherwise, to dynamically set and change the blend constant, call:
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)
- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `commandBuffer` must be in the recording state
- 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>

### 26.1.2. 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.

### 26.1.3. 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:

```c
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,
} VkBlendOp;
```
The semantics of each basic blend operations is 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.
If the framebuffer color attachment numeric format is not sRGB encoded then the resulting color 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. Framebuffer color attachments greater than or equal to `VkSubpassDescription::colorAttachmentCount` perform no writes.

26.2. 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`. 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:

```c
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:

- ¬ is bitwise invert,
- ∧ is bitwise and,
- ∨ is bitwise or,
- ⊕ is bitwise exclusive or,
- s is the fragment’s R<sub>s0</sub>, G<sub>s0</sub>, B<sub>s0</sub> or A<sub>s0</sub> 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 ∧ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_REVERSE</td>
<td>s ∧ ¬ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY</td>
<td>s</td>
</tr>
<tr>
<td>VK_LOGIC_OP_AND_INVERTED</td>
<td>¬ s ∧ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NO_OP</td>
<td>d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_XOR</td>
<td>s ⊕ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR</td>
<td>s ∨ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NOR</td>
<td>¬ (s ∨ d)</td>
</tr>
<tr>
<td>VK_LOGIC_OP_EQUVALENT</td>
<td>¬ (s ⊕ d)</td>
</tr>
<tr>
<td>VK_LOGIC_OP_INVERT</td>
<td>¬ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_REVERSE</td>
<td>s ∨ ¬ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_COPY_INVERTED</td>
<td>¬ s</td>
</tr>
<tr>
<td>VK_LOGIC_OP_OR_INVERTED</td>
<td>¬ s ∨ d</td>
</tr>
<tr>
<td>VK_LOGIC_OP_NAND</td>
<td>¬ (s ∧ 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.

### 26.3. 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.

typedef VkFlags VkColorComponentFlags;

*VkColorComponentFlags* is a bitmask type for setting a mask of zero or more *VkColorComponentFlagBits*.

Chapter 27. 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
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` × `groupCountY` × `groupCountZ` local workgroups is assembled.
Valid Usage

- If a VkImageView is sampled with VK_FILTER_LINEAR as a result of this command, then the image view's format features must contain VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT.

- 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.

- 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.

- 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 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.

- `groupCountX` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]`
- `groupCountY` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]`
- `groupCountZ` must be less than or equal to `VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]`

Valid Usage (Implicit)

- `commandBuffer` must be a valid `VkCommandBuffer` handle
- `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>Compute</td>
</tr>
<tr>
<td>Secondary</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To record an indirect command dispatch, call:

```c
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

- If a \texttt{VkImageView} is sampled with \texttt{VK_FILTER_LINEAR} as a result of this command, then the image view's format features \textbf{must} contain \texttt{VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT}

- If a \texttt{VkImageView} is accessed using atomic operations as a result of this command, then the image view's format features \textbf{must} contain \texttt{VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT}

- For each set \( n \) that is statically used by the \texttt{VkPipeline} bound to the pipeline bind point used by this command, a descriptor set \textbf{must} have been bound to \( n \) at the same pipeline bind point, with a \texttt{VkPipelineLayout} that is compatible for set \( n \), with the \texttt{VkPipelineLayout} used to create the current \texttt{VkPipeline}, as described in Pipeline Layout Compatibility

- For each push constant that is statically used by the \texttt{VkPipeline} bound to the pipeline bind point used by this command, a push constant value \textbf{must} have been set for the same pipeline bind point, with a \texttt{VkPipelineLayout} that is compatible for push constants, with the \texttt{VkPipelineLayout} used to create the current \texttt{VkPipeline}, as described in Pipeline Layout Compatibility

- 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

- A valid pipeline \textbf{must} be bound to the pipeline bind point used by this command

- 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 for \texttt{commandBuffer}, and done so after any previously bound pipeline with the corresponding state not specified as dynamic

- 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

- 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

- 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

- 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

- 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
• 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 **buffer** is non-sparse then it **must** be bound completely and contiguously to a single **VkDeviceMemory** object.

• **buffer** **must** have been created with the **VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT** bit set.

• **offset** **must** be a multiple of **4**

• The sum of **offset** and the size of **VkDispatchIndirectCommand** **must** be less than or equal to the size of **buffer**

---

### Valid Usage (Implicit)

- **commandBuffer** **must** be a valid **VkCommandBuffer** handle.
- **buffer** **must** be a valid **VkBuffer** handle.
- **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.
- 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 \texttt{VkDispatchIndirectCommand} have the same meaning as the corresponding parameters of \texttt{vkCmdDispatch}.

### Valid Usage

- \( x \) must be less than or equal to \texttt{VkPhysicalDeviceLimits::maxComputeWorkGroupCount[0]}
- \( y \) must be less than or equal to \texttt{VkPhysicalDeviceLimits::maxComputeWorkGroupCount[1]}
- \( z \) must be less than or equal to \texttt{VkPhysicalDeviceLimits::maxComputeWorkGroupCount[2]}
Chapter 28. 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.

28.1. 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_PREFERABLE_RESIDENCY_BIT` and `VK_BUFFER_CREATE_PREFERABLE_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_PREFERABLE_RESIDENCY_BIT`.
  - `sparseResidencyImage2D`: Support for creating 2D single-sampled `VkImage` objects with `VK_IMAGE_CREATE_PREFERABLE_RESIDENCY_BIT`.
  - `sparseResidencyImage3D`: Support for creating 3D `VkImage` objects with `VK_IMAGE_CREATE_PREFERABLE_RESIDENCY_BIT`.
  - `sparseResidency2Samples`: Support for creating 2D `VkImage` objects with 2 samples and `VK_IMAGE_CREATE_PREFERABLE_RESIDENCY_BIT`.
  - `sparseResidency4Samples`: Support for creating 2D `VkImage` objects with 4 samples and `VK_IMAGE_CREATE_PREFERABLE_RESIDENCY_BIT`.
  - `sparseResidency8Samples`: Support for creating 2D `VkImage` objects with 8 samples and `VK_IMAGE_CREATE_PREFERABLE_RESIDENCY_BIT`.
  - `sparseResidency16Samples`: Support for creating 2D `VkImage` objects with 16 samples and `VK_IMAGE_CREATE_PREFERABLE_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_PREFERABLE_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.

### 28.2. Sparse Buffers and Fully-Resident Images

Both `VkBuffer` and `VkImage` objects created with the `VK_IMAGE_CREATE_PREFERABLE_BINDING_BIT` or `VK_BUFFER_CREATE_PREFERABLE_BINDING_BIT` bits can be thought of as a linear region of address space. In the `VkImage` case if `VK_IMAGE_CREATE_PREFERABLE_RESIDENCY_BIT` is not used, this linear region is entirely opaque, meaning that there is no application-visible mapping between texel location and memory...
offset.

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.

### 28.2.1. 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.

### 28.3. 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.

### 28.4. 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.

### 28.4.1. 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.

28.4.2. 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.

![Diagram](image)

**Figure 9. Sparse Image**

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 \texttt{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.

![Sparse Image with Aligned Mip Size and Single Mip Tail](image)

\textit{Figure 12. Sparse Image with Aligned Mip Size and Single Mip Tail}

\textit{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 \texttt{VK\_SPARSE\_IMAGE\_FORMAT\_ALIGNED\_MIP\_SIZE\_BIT} and \texttt{VK\_SPARSE\_IMAGE\_FORMAT\_SINGLE\_MIPTAIL\_BIT} are present the constraints from each of these flags are in effect.

\subsection{28.4.3. 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. \texttt{VK\_FORMAT\_BC5\_UNORM\_BLOCK}), the texel size is the size of the compressed texel block (e.g. 128-bit for \texttt{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 27. 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 28. 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.

### 28.4.4. 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.

28.4.5. 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 13. 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 ¼ the size of the depth data, hence the stencil sparse blocks include 4 × 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.

### 28.5. 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.
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) “dummy” 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 dummy 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 → 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 values makes it easiest for the implementation to derive the correct device virtual address.

28.7. 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
28.7.1. 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.

28.7.2. 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
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 were populated with 0; writes to non-resident regions will be discarded.

### 28.7.3. 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:

```c
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.

```c
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 capabilities.
- **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.
- **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**

- **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)

- `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `format` must be a valid `VkFormat` value
- `type` must be a valid `VkImageType` value
- `samples` must be a valid `VkSampleCountFlagBits` value
- `usage` must be a valid combination of `VkImageUsageFlagBits` values
- `usage` must not be 0
- `tiling` must be a valid `VkImageTiling` value
- `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

28.7.4. 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) at CreateDevice time. 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`).

28.7.5. 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
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:

```c
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)

- **device** must be a valid `VkDevice` handle
- **image** must be a valid `VkImage` handle
- `pSparseMemoryRequirementCount` must be a valid pointer to a `uint32_t` value
- 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
- **image** must have been created, allocated, or retrieved from `device`

28.7.6. 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.

**Note**

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:

$$\text{arrayMipTailOffset} = \text{imageMipTailOffset} + \text{arrayLayer} \times \text{imageMipTailStride};$$

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
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:

$$\text{metadataRegion} = [\text{base}, \text{base} + \text{imageMipTailSize})$$
base = \text{imageMipTailOffset} + \text{imageMipTailStride} \times n

and \text{imageMipTailOffset}, \text{imageMipTailSize}, \text{and} \text{imageMipTailStride} \text{ values are from the } \text{VkSparseImageMemoryRequirements} \text{ corresponding to the metadata aspect of the image, and } n \text{ is a valid array layer index for the image, }

\text{imageMipTailStride} \text{ is considered to be zero for aspects where } \text{VkSparseImageMemoryRequirements} ::\text{formatProperties.flags} \text{ contains } \text{VK\_SPARSE\_IMAGE\_FORMAT\_SINGLE\_MIPTAIL\_BIT}.

If flags does not contain \text{VK\_SPARSE\_MEMORY\_BIND\_METADATA\_BIT}, the binding range \textbf{must} be within the range \( [0, \text{VkMemoryRequirements}::\text{size}) \).

---

### Valid Usage

- If memory is not \text{VK\_NULL\_HANDLE}, memory and memoryOffset \textbf{must} match the memory requirements of the resource, as described in section Resource Memory Association
- If memory is not \text{VK\_NULL\_HANDLE}, memory \textbf{must} not have been created with a memory type that reports \text{VK\_MEMORY\_PROPERTY\_LAZYLY\_ALLOCATED\_BIT} \text{ bit set}
- size \textbf{must} be greater than 0
- resourceOffset \textbf{must} be less than the size of the resource
- size \textbf{must} be less than or equal to the size of the resource minus resourceOffset
- memoryOffset \textbf{must} be less than the size of memory
- size \textbf{must} be less than or equal to the size of memory minus memoryOffset

---

### Valid Usage (Implicit)

- If memory is not \text{VK\_NULL\_HANDLE}, memory \textbf{must} be a valid \text{VkDeviceMemory} handle
- flags \textbf{must} be a valid combination of \text{VkSparseMemoryBindFlagBits} values

---

Bits which \textbf{can} be set in \text{VkSparseMemoryBind}::flags, specifying usage of a sparse memory binding operation, are:

```cpp
typedef enum VkSparseMemoryBindFlagBits {
    VK\_SPARSE\_MEMORY\_BIND\_METADATA\_BIT = 0x00000001,
} VkSparseMemoryBindFlagBits;
```

- \text{VK\_SPARSE\_MEMORY\_BIND\_METADATA\_BIT} specifies that the memory being bound is only for the metadata aspect.

```cpp
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:

```c
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 array of VkSparseMemoryBind structures.

**Valid Usage (Implicit)**

- `buffer` must be a valid VkBuffer handle
- `pBinds` must be a valid pointer to an array of `bindCount` valid VkSparseMemoryBind structures
- `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:

```c
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**

- 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)

- **image** must be a valid `VkImage` handle
- **pBinds** must be a valid pointer to an array of `bindCount` valid `VkSparseMemoryBind` structures
- **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
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

• The `subresource.mipLevel` member of each element of `pBinds` must be less than the `mipLevels` specified in `VkImageCreateInfo` when `image` was created

• The `subresource.arrayLayer` member of each element of `pBinds` must be less than the `arrayLayers` specified in `VkImageCreateInfo` when `image` was created

• `image` must have been created with `VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT` set

Valid Usage (Implicit)

• `image` must be a valid `VkImage` handle

• `pBinds` must be a valid pointer to an array of `bindCount` valid `VkSparseImageMemoryBind` structures

• `bindCount` must be greater than 0

The `VkSparseImageMemoryBind` structure is defined as:

```c
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

• 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

• memory and memoryOffset must match the memory requirements of the calling command’s image, as described in section Resource Memory Association

• subresource must be a valid image subresource for image (see Image Views)

• offset.x must be a multiple of the sparse image block width (VkSparseImageFormatProperties::imageGranularity.width) of the image

• extent.width must either be a multiple of the sparse image block width of the image, or else (extent.width + offset.x) must equal the width of the image subresource

• offset.y must be a multiple of the sparse image block height (VkSparseImageFormatProperties::imageGranularity.height) of the image

• extent.height must either be a multiple of the sparse image block height of the image, or else (extent.height + offset.y) must equal the height of the image subresource

• offset.z must be a multiple of the sparse image block depth (VkSparseImageFormatProperties::imageGranularity.depth) of the image

• extent.depth must either be a multiple of the sparse image block depth of the image, or else (extent.depth + offset.z) must equal the depth of the image subresource

Valid Usage (Implicit)

• subresource must be a valid VkImageSubresource structure

• If memory is not VK_NULL_HANDLE, memory must be a valid VkDeviceMemory handle

• flags must be a valid combination of VkSparseMemoryBindFlagBits values

To submit sparse binding operations to a queue, call:

```
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.

\texttt{vkQueueBindSparse} is a queue submission command, with each batch defined by an element of \texttt{pBindInfo} as a \texttt{VkBindSparseInfo} structure. Batches begin execution in the order they appear in \texttt{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 \texttt{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

- If \texttt{fence} is not \texttt{VK_NULL_HANDLE}, \texttt{fence} must be unsignaled
- If \texttt{fence} is not \texttt{VK_NULL_HANDLE}, \texttt{fence} must not be associated with any other queue command that has not yet completed execution on that queue
- Each element of the \texttt{pSignalSemaphores} member of each element of \texttt{pBindInfo} must be unsignaled when the semaphore signal operation it defines is executed on the device
- When a semaphore wait operation referring to a binary semaphore defined by any element of the \texttt{pWaitSemaphores} member of any element of \texttt{pBindInfo} executes on \texttt{queue}, there must be no other queues waiting on the same semaphore
- All elements of the \texttt{pWaitSemaphores} member of all elements of \texttt{pBindInfo} member referring to a binary semaphore must be semaphores that are signaled, or have semaphore signal operations previously submitted for execution

### Valid Usage (Implicit)

- \texttt{queue} must be a valid \texttt{VkQueue} handle
- If \texttt{bindInfoCount} is not \texttt{0}, \texttt{pBindInfo} must be a valid pointer to an array of \texttt{bindInfoCount} valid \texttt{VkBindSparseInfo} structures
- If \texttt{fence} is not \texttt{VK_NULL_HANDLE}, \texttt{fence} must be a valid \texttt{VkFence} handle
- The \texttt{queue} must support sparse binding operations
- 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 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:

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 an extension-specific 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 (Implicit)**

• **sType** must be **VK_STRUCTURE_TYPE_BIND_SPARSE_INFO**

• **pNext** must be NULL

• If **waitSemaphoreCount** is not 0, **pWaitSemaphores** must be a valid pointer to an array of **waitSemaphoreCount** valid VkSemaphore handles

• If **bufferBindCount** is not 0, **pBufferBinds** must be a valid pointer to an array of **bufferBindCount** valid VkSparseBufferMemoryBindInfo structures

• If **imageOpaqueBindCount** is not 0, **pImageOpaqueBinds** must be a valid pointer to an array of **imageOpaqueBindCount** valid VkSparseImageOpaqueMemoryBindInfo structures

• If **imageBindCount** is not 0, **pImageBinds** must be a valid pointer to an array of **imageBindCount** valid VkSparseImageMemoryBindInfo structures

• If **signalSemaphoreCount** is not 0, **pSignalSemaphores** must be a valid pointer to an array of **signalSemaphoreCount** valid VkSemaphore handles

• 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
28.8. Examples

The following examples illustrate basic creation of sparse images and binding them to physical memory.

28.8.1. Basic Sparse Resources

This basic example creates a normal VkImage object but uses fine-grained memory allocation to back the resource with multiple memory ranges.

```c
VkDevice                device;
VkQueue                 queue;
VkImage                 sparseImage;
VkAllocationCallbacks*  pAllocator = NULL;
VkMemoryRequirements    memoryRequirements = {};
VkDeviceSize            offset = 0;
VkSparseMemoryBind      binds[MAX_CHUNKS] = {}; // MAX_CHUNKS is NOT part of Vulkan
uint32_t                bindCount = 0;

// Allocate image object
const VkImageCreateInfo sparseImageInfo = {
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO, // sType
    NULL, // pNext
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT | ..., // flags
    ...}
vkCreateImage(device, &sparseImageInfo, pAllocator, &sparseImage);

// Get memory requirements
vkGetImageMemoryRequirements(
    device,
    sparseImage,
    &memoryRequirements);

// Bind memory in fine-grained fashion, find available memory ranges
// from potentially multiple VkDeviceMemory pools.
// (Illustration purposes only, can be optimized for perf)
while (memoryRequirements.size && bindCount < MAX_CHUNKS)
{
    VkSparseMemoryBind* pBind = &binds[bindCount];
    pBind->resourceOffset = offset;
    AllocateOrGetMemoryRange(
        device,
        &memoryRequirements,
        &pBind->memory,
        &pBind->memoryOffset,
        ...}
```

590 | Chapter 28. Sparse Resources
// memory ranges must be sized as multiples of the alignment
assert(IsMultiple(pBind->size, memoryRequirements.alignment));
assert(IsMultiple(pBind->memoryOffset, memoryRequirements.alignment));

memoryRequirements.size -= pBind->size;
offset += pBind->size;
bindCount++;

// Ensure all image has backing
if (memoryRequirements.size)
{
    // Error condition - too many chunks
}

const VkSparseImageOpaqueMemoryBindInfo opaqueBindInfo =
{
    sparseImage, // image
   (bindCount, // bindCount
    binds       // pBinds
};

const VkBindSparseInfo bindSparseInfo =
{
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO, // sType
    NULL, // pNext
    ...,
    1,    // imageOpaqueBindCount
    &opaqueBindInfo, // pImageOpaqueBinds
    ...,
};

// vkQueueBindSparse is externally synchronized per queue object.
AcquireQueueOwnership(queue);

// Actually bind memory
vkQueueBindSparse(queue, 1, &bindSparseInfo, VK_NULL_HANDLE);

ReleaseQueueOwnership(queue);

### 28.8.2. Advanced Sparse Resources

This more advanced example creates an arrayed color attachment / texture image and binds only
LOD zero and the **required** metadata to physical memory.

```
VkDevice                            device;
VkQueue                             queue;
VkImage                             sparseImage;
```
VkAllocationCallbacks* pAllocator = NULL;
VkMemoryRequirements memoryRequirements = {0};
uint32_t sparseRequirementsCount = 0;
VkSparseImageMemoryRequirements* pSparseReqs = NULL;
VkSparseMemoryBind binds[MY_IMAGE_ARRAY_SIZE] = {0};
VkSparseImageMemoryBind imageBinds[MY_IMAGE_ARRAY_SIZE] = {0};
uint32_t bindCount = 0;

// Allocate image object (both renderable and sampleable)
const VkImageCreateInfo sparseImageInfo = {
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO, // sType
    NULL, // pNext
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT | ... // flags
    ... // format
    ... // arrayLayers
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT, // usage
    ... // device
};
vkCreateImage(device, &sparseImageInfo, pAllocator, &sparseImage);

// Get memory requirements
vkGetImageMemoryRequirements(
    device,
    sparseImage,
    &memoryRequirements);

// Get sparse image aspect properties
vkGetImageSparseMemoryRequirements(
    device,
    sparseImage,
    &sparseRequirementsCount,
    NULL);

pSparseReqs = (VkSparseImageMemoryRequirements*)
    malloc(sparseRequirementsCount * sizeof(VkSparseImageMemoryRequirements));

vkGetImageSparseMemoryRequirements(
    device,
    sparseImage,
    &sparseRequirementsCount,
    pSparseReqs);

// Bind LOD level 0 and any required metadata to memory
for (uint32_t i = 0; i < sparseRequirementsCount; ++i)
{
    if (pSparseReqs[i].formatProperties.aspectMask &
assert(pSparseReqs[i].formatProperties.aspectMask == VK_IMAGE_ASPECT_METADATA_BIT);

if (pSparseReqs[i].formatProperties.flags & VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT)
{
    VkSparseMemoryBind* pBind = &binds[bindCount];
    pBind->memorySize = pSparseReqs[i].imageMipTailSize;
    bindCount++;

    // ... Allocate memory range

    pBind->resourceOffset = pSparseReqs[i].imageMipTailOffset;
    pBind->memoryOffset = /* allocated memoryOffset */;
    pBind->memory = /* allocated memory */;
    pBind->flags = VK_SPARSE_MEMORY_BIND_METADATA_BIT;
}
else
{
    // Need a mip tail region per array layer.
    for (uint32_t a = 0; a < sparseImageInfo.arrayLayers; ++a)
    {
        VkSparseMemoryBind* pBind = &binds[bindCount];
        pBind->memorySize = pSparseReqs[i].imageMipTailSize;
        bindCount++;

        // ... Allocate memory range

        pBind->resourceOffset = pSparseReqs[i].imageMipTailOffset +
            (a * pSparseReqs[i].imageMipTailStride);

        pBind->memoryOffset = /* allocated memoryOffset */;
        pBind->memory = /* allocated memory */;
        pBind->flags = VK_SPARSE_MEMORY_BIND_METADATA_BIT;
    }
}
else
{
    // resource data

    VkExtent3D lod0BlockSize = {
    }
    
    AlignedDivide( 
        sparseImageInfo.extent.width, 
        pSparseReqs[i].formatProperties.imageGranularity.width);
    AlignedDivide( 
        sparseImageInfo.extent.height, 
        pSparseReqs[i].formatProperties.imageGranularity.height);
size_t totalBlocks =
    lod0BlockSize.width * 
    lod0BlockSize.height * 
    lod0BlockSize.depth;

// Each block is the same size as the alignment requirement, 
// calculate total memory size for level 0
VkDeviceSize lod0MemSize = totalBlocks * memoryRequirements.alignment;

// Allocate memory for each array layer
for (uint32_t a = 0; a < sparseImageInfo.arrayLayers; ++a)
{
    // ... Allocate memory range

    VkSparseImageMemoryBind* pBind = &imageBinds[a];
    pBind->subresource.aspectMask = pSparseReqs[i].formatProperties.aspectMask;
    pBind->subresource.mipLevel = 0;
    pBind->subresource.arrayLayer = a;

    pBind->offset = (VkOffset3D){0, 0, 0};
    pBind->extent = sparseImageInfo.extent;
    pBind->memoryOffset = /* allocated memoryOffset */;
    pBind->memory = /* allocated memory */;
    pBind->flags = 0;
}

free(pSparseReqs);
}

const VkSparseImageOpaqueMemoryBindInfo opaqueBindInfo =
{
    sparseImage,  // image
    bindCount,   // bindCount
    binds        // pBinds
};

const VkSparseImageMemoryBindInfo imageBindInfo =
{
    sparseImage,  // image
    sparseImageInfo.arrayLayers, // bindCount
    imageBinds    // pBinds
};

const VkBindSparseInfo bindSparseInfo =
{  
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO,  // sType
    NULL,  // pNext
    
    1,  // imageOpaqueBindCount
    &opaqueBindInfo,  // pImageOpaqueBinds
    1,  // imageBindCount
    &imageBindInfo,  // pImageBinds
    ...
};

// vkQueueBindSparse is externally synchronized per queue object.
AcquireQueueOwnership(queue);

// Actually bind memory
vkQueueBindSparse(queue, 1, &bindSparseInfo, VK_NULL_HANDLE);

ReleaseQueueOwnership(queue);
Chapter 29. 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.

29.1. 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 accept a `VkDevice` object or any of a device's child objects as a parameter, are considered device-level functionality.

29.2. 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.

29.2.1. Version Numbers

The Vulkan version number comprises three parts indicating the major, minor and patch version of the Vulkan API Specification.

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 major version is a 10-bit integer packed into bits 31-22.
- 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_VERSION_MAJOR` extracts the API major version number from a packed version number:
```c
#define VK_VERSION_MAJOR(version) ((uint32_t)(version) >> 22)

VK_VERSION_MINOR extracts the API minor version number from a packed version number:

#define VK_VERSION_MINOR(version) (((uint32_t)(version) >> 12) & 0x3ff)

VK_VERSION_PATCH extracts the API patch version number from a packed version number:

#define VK_VERSION_PATCH(version) ((uint32_t)(version) & 0xfff)

VK_MAKE_VERSION constructs an API version number.

#define VK_MAKE_VERSION(major, minor, patch) 
  (((major) << 22) | ((minor) << 12) | (patch))

- `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.

// Vulkan 1.0 version number
#define VK_API_VERSION_1_0 VK_MAKE_VERSION(1, 0, 0) // Patch version should always be set to 0
```

### 29.2.2. Querying Version Support

**Note**

In Vulkan 1.0, there is no mechanism to detect the separate versions of instance-level and device-level functionality supported. However, the `vkEnumerateInstanceVersion` command was added in Vulkan 1.1 to determine the supported version of instance-level functionality - querying for this function via `vkGetInstanceProcAddr` will return `NULL` on implementations that only support Vulkan 1.0 functionality. For more information on this, please refer to the Vulkan 1.1 specification.

The version of device-level functionality can be queried by calling `vkGetPhysicalDeviceProperties` and is returned in `VkPhysicalDeviceProperties::apiVersion`, encoded as described in **Version Numbers**.
29.3. 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:

```plaintext
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. If `pPropertyCount` is smaller than the number of layers available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available layer 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)

- `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` `VkLayerProperties` structures

Return Codes

Success
- `VK_SUCCESS`
- `VK_INCOMPLETE`

Failure
- `VK_ERROR_OUT_OF_HOST_MEMORY`
- `VK_ERROR_OUT_OF_DEVICE_MEMORY`

The `VkLayerProperties` structure is defined as:

```c
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.

### 29.3.1. 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
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. If `pPropertyCount` is smaller than the number of layers available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available layer 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)

- **physicalDevice** must be a valid `VkPhysicalDevice` handle.
- **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` `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.

### 29.4. 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.

Enabling an 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.

29.4.1. 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
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 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 extension properties available, at most pPropertyCount structures will be written. If pPropertyCount is smaller than the number of extensions available, VK_INCOMPLETE will be returned instead of VK_SUCCESS, to indicate that not all the available properties were returned.
Because the list of available layers may change externally between calls to `vkEnumerateInstanceExtensionProperties`, two calls may retrieve different results if a `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 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)

- If `pLayerName` is not NULL, `pLayerName` must be a null-terminated UTF-8 string
- `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` `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`

### 29.4.2. Device Extensions

Device extensions add new device-level functionality to the API, outside of the core specification.

To query the extensions available to a given physical device, call:

```c
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)

- `physicalDevice` **must** be a valid `VkPhysicalDevice` handle
- If `pLayerName` is not `NULL`, `pLayerName` **must** be a null-terminated UTF-8 string
- `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` `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
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.
29.5. Extension Dependencies

Some extensions are dependent on other extensions to function. To enable extensions with dependencies, such 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 vk.xml describing some aspects of promotion, especially requires, promotedto and deprecatedby attributes of <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.

29.6. 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.

29.6.1. 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. 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 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.

29.6.2. 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` 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.
Chapter 30. 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.

For convenience, new core versions of Vulkan *may* introduce new unified features structures for features promoted from extensions. At the same time, the extension’s original features 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 29. Extension Feature Aliases**

<table>
<thead>
<tr>
<th>Extension</th>
<th>Feature(s)</th>
</tr>
</thead>
</table>

To query supported features, call:

```c
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)**

- `physicalDevice` *must* be a valid `VkPhysicalDevice` handle
- `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.
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` is equivalent to setting all members of the structure to `VK_FALSE`.

Note

Some features, such as `robustBufferAccess`, may incur a run-time performance cost. Application writers should carefully consider the implications of enabling all supported features.

The `VkPhysicalDeviceFeatures` structure is defined as:

```c
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;
} VkPhysicalDeviceFeatures;
```
The members of the `VkPhysicalDeviceFeatures` structure describe 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.

    ```
    Note
    If a SPIR-V `OpLoad` instruction loads a structure and the tail end of the structure is out of bounds, then all members of the structure are considered out of bounds even if the members at the end are not statically used.
    ```

    - If any buffer access is determined to be out of bounds, then any other access of the same type (load, store, or atomic) to the same buffer that accesses an address less than 16 bytes away from the out of bounds address may also be considered out of bounds.

    - If the access is a load that reads from the same memory locations as a prior store in the same shader invocation, with no other intervening accesses to the same memory locations in that shader invocation, then the result of the load may be the value stored
by the store instruction, even if the access is out of bounds. If the load is \texttt{Volatile}, then
an out of bounds load \textbf{must} return the appropriate out of bounds value.

\begin{itemize}
\item Out-of-bounds buffer loads will return any of the following values:
  \begin{itemize}
  \item 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).
  \item Zero values, or \((0,0,0,\ldots,0)\) vectors for vector reads where \(x\) is a valid value represented in the type of the vector components and \textbf{may} be any of:
    \begin{itemize}
    \item 0, 1, or the maximum representable positive integer value, for signed or unsigned integer components
    \item 0.0 or 1.0, for floating-point components
    \end{itemize}
  \end{itemize}
\end{itemize}

\begin{itemize}
\item Out-of-bounds writes \textbf{may} modify values within the memory range(s) bound to the buffer, but \textbf{must} not modify any other memory.
\item Out-of-bounds atomics \textbf{may} modify values within the memory range(s) bound to the buffer, but \textbf{must} not modify any other memory, and return an undefined value.
\item 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:
  \begin{itemize}
  \item \texttt{vertexBufferRangeSize}, if \texttt{bindingStride} == 0; or
  \item \((\texttt{vertexBufferRangeSize} - (\texttt{vertexBufferRangeSize} \% \texttt{bindingStride}))\)
  \end{itemize}
\end{itemize}

where \texttt{vertexBufferRangeSize} is the byte size of the memory range bound to the vertex buffer binding and \texttt{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.

\begin{itemize}
\item If a vertex input attribute is out of bounds, it will be assigned one of the following values:
  \begin{itemize}
  \item Values from anywhere within the memory range(s) bound to the buffer, converted according to the format of the attribute.
  \item Zero values, format converted according to the format of the attribute.
  \item Zero values, or \((0,0,0,\ldots,0)\) vectors, as described above.
  \end{itemize}
\end{itemize}

\begin{itemize}
\item If \texttt{robustBufferAccess} is not enabled, applications \textbf{must} not perform out of bounds accesses.
\end{itemize}

\begin{itemize}
\item \texttt{fullDrawIndexUint32} specifies the full 32-bit range of indices is supported for indexed draw calls when using a \texttt{VkIndexType} of \texttt{VK_INDEX_TYPE_UINT32}. \texttt{maxDrawIndexedIndexValue} is the maximum index value that \textbf{may} be used (aside from the primitive restart index, which is always \(2^{32}-1\) when the \texttt{VkIndexType} is \texttt{VK_INDEX_TYPE_UINT32}). If this feature is supported, \texttt{maxDrawIndexedIndexValue} \textbf{must} be \(2^{32}-1\); otherwise it \textbf{must} be no smaller than \(2^{24}-1\). See \texttt{maxDrawIndexedIndexValue}.
\item \texttt{imageCubeArray} specifies whether image views with a \texttt{VkImageViewType} of \texttt{VK_IMAGE_VIEW_TYPE_CUBE_ARRAY} \textbf{can} be created, and that the corresponding \texttt{SampledCubeArray} and \texttt{ImageCubeArray} SPIR-V capabilities \textbf{can} be used in shader code.
\item \texttt{independentBlend} specifies whether the \texttt{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 `VkPhysicalDeviceLimits` 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 color fragment output from the fragment shader with the maximum representable alpha value for fixed-point colors or 1.0 for floating-point colors. 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.

• **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`
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 \texttt{VKQUERY\_TYPE\_PIPELINE\_STATISTICS} cannot be created, and none of the \texttt{VkQueryPipelineStatisticFlagBits} bits can be set in the \texttt{pipelineStatistics} member of the \texttt{VkQueryPoolCreateInfo} structure.

- \texttt{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 buffers, and storage buffer variables used by these stages in shader modules must be decorated with the \texttt{NonWritable} decoration (or the \texttt{readonly} memory qualifier in GLSL).

- \texttt{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 buffers, and storage buffer variables used by the fragment stage in shader modules must be decorated with the \texttt{NonWritable} decoration (or the \texttt{readonly} memory qualifier in GLSL).

- \texttt{shaderTessellationAndGeometryPointSize} specifies whether the \texttt{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 \texttt{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 \texttt{TessellationPointSize} capability for tessellation control and evaluation shaders, or if the shader modules can declare the \texttt{GeometryPointSize} capability for geometry shaders. An implementation supporting this feature must also support one or both of the \texttt{tessellationShader} or \texttt{geometryShader} features.

- \texttt{shaderImageGatherExtended} specifies whether the extended set of image gather instructions are available in shader code. If this feature is not enabled, the \texttt{OpImage*Gather} instructions do not support the \texttt{Offset} and \texttt{ConstOffsets} operands. This also specifies whether shader modules can declare the \texttt{ImageGatherExtended} capability.

- \texttt{shaderStorageImageExtendedFormats} specifies whether all the “storage image extended formats” below are supported; if this feature is supported, then the \texttt{VK\_FORMAT\_FEATURE\_STORAGE\_IMAGE\_BIT} must be supported in \texttt{optimalTilingFeatures} for the following formats:

  - \texttt{VK\_FORMAT\_R16G16\_SFLOAT}
  - \texttt{VK\_FORMAT\_B10G11R11\_UFLOAT\_PACK32}
  - \texttt{VK\_FORMAT\_R16\_SFLOAT}
  - \texttt{VK\_FORMAT\_R16G16B16\_A16\_UNORM}
  - \texttt{VK\_FORMAT\_A2B10G10R10\_UNORM\_PACK32}
  - \texttt{VK\_FORMAT\_R16G16\_UNORM}
  - \texttt{VK\_FORMAT\_R8G8\_UNORM}
  - \texttt{VK\_FORMAT\_R16\_UNORM}
  - \texttt{VK\_FORMAT\_R8\_UNORM}
  - \texttt{VK\_FORMAT\_R16G16B16\_A16\_SNORM}
  - \texttt{VK\_FORMAT\_R16G16\_SNORM}
  - \texttt{VK\_FORMAT\_R8G8\_SNORM}
Note

The 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` capability.

- `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`, 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. If 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.

nullDescriptor support requires the VK_EXT_robustness2 extension.

30.1. Feature Requirements

All Vulkan graphics implementations must support the following features:

• robustBufferAccess

All other features defined in the Specification are optional.
Chapter 31. Limits

Limits are implementation-dependent minimums, maximums, and other device characteristics that an application may need to be aware of.

The VkPhysicalDeviceLimits structure is defined as:

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 maxPerStageResources;
    uint32_t maxDescriptorSetSamplers;
    uint32_t maxDescriptorSetUniformBuffers;
    uint32_t maxDescriptorSetUniformBuffersDynamic;
    uint32_t maxDescriptorSetStorageBuffers;
    uint32_t maxDescriptorSetStorageBuffersDynamic;
    uint32_t maxDescriptorSetSampledImages;
    uint32_t maxDescriptorSetStorageImages;
    uint32_t maxDescriptorSetInputAttachments;
    uint32_t maxVertexInputAttributes;
    uint32_t maxVertexInputBindings;
    uint32_t maxVertexInputAttributeOffset;
    uint32_t maxVertexInputBindingStride;
    uint32_t maxVertexOutputComponents;
    uint32_t maxTessellationGenerationLevel;
    uint32_t maxTessellationPatchSize;
    uint32_t maxTessellationControlPerVertexInputComponents;
    uint32_t maxTessellationControlPerVertexOutputComponents;
    uint32_t maxTessellationControlPerPatchOutputComponents;
    uint32_t maxTessellationControlTotalOutputComponents;
    uint32_t maxTessellationEvaluationInputComponents;
    uint32_t maxTessellationEvaluationOutputComponents;
} VkPhysicalDeviceLimits;
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  maxGeometryTotalOutputComponents;
uint32_t  maxGeometryTotalOutputComponents;
uint32_t  maxFragmentInputComponents;
uint32_t  maxFragmentOutputAttachments;
uint32_t  maxFragmentDualSrcAttachments;
uint32_t  maxFragmentCombinedOutputResources;
uint32_t  maxGeometryTotalOutputComponents;
uint32_t  maxGeometryTotalOutputComponents;
uint32_t  maxFragmentInputComponents;
uint32_t  maxFragmentOutputAttachments;
uint32_t  maxFragmentCombinedOutputResources;
uint32_t  subPixelPrecisionBits;
uint32_t  subTexelPrecisionBits;
mipmapPrecisionBits;
uint32_t  maxDrawIndexedIndexValue;
uint32_t  maxDrawIndirectCount;
float    maxSamplerLodBias;
float    maxSamplerAnisotropy;
uint32_t  maxViewports;
uint32_t  maxViewportDimensions;
float    maxViewportDimensions;
viewportDimensions;
float    viewportBoundsRange;
viewportDimensions;
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;
VkSampleCountFlags framebufferNoAttachmentsSampleCounts;
uint32_t  maxColorAttachments;
VkSampleCountFlags sampledImageColorSampleCounts;
VkSampleCountFlags sampledImageIntegerSampleCounts;
VkSampleCountFlags sampledImageDepthSampleCounts;
VkSampleCountFlags sampledImageStencilSampleCounts;
VkSampleCountFlags storageImageSampleCounts;
uint32_t  maxSampleMaskWords;
VkBool32   timestampComputeAndGraphics;
float2      timestampPeriod;
uint32_t              maxClipDistances;
uint32_t              maxCullDistances;
uint32_t              maxCombinedClipAndCullDistances;
uint32_t              discreteQueuePriorities;
float                  pointSizeRange[2];
float                  lineWidthRange[2];
float                  pointSizeGranularity;
float                  lineWidthGranularity;
VkBool32              strictLines;
VkBool32              standardSampleLocations;
VkDeviceSize          optimalBufferCopyOffsetAlignment;
VkDeviceSize          optimalBufferCopyRowPitchAlignment;
VkDeviceSize          nonCoherentAtomSize;
} VkPhysicalDeviceLimits;

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 maximum dimension (width) supported for all images created with an imageType of VK_IMAGE_TYPE_1D.
- **maxImageDimension2D** is the maximum dimension (width or height) supported for all images created with an imageType of VK_IMAGE_TYPE_2D and without VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT set in flags.
- **maxImageDimension3D** is the maximum dimension (width, height, or depth) supported for all images created with an imageType of VK_IMAGE_TYPE_3D.
- **maxImageDimensionCube** is the maximum dimension (width or height) supported for all images created with an imageType of VK_IMAGE_TYPE_2D and with VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT set in flags.
- **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 any VkDescriptorBufferInfo structures passed to a call 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 any VkDescriptorBufferInfo structures passed to a call 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 size 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. 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. 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. 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. 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. 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. 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 Input Attachment.
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_STORAGE_TEXEL_BUFFER`, `VK_DESCRIPTOR_TYPE_UNIFORM_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. 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. See Sampler and Combined Image Sampler.

- **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. 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. 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. 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. 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. 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. 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. 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 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, 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 buffers 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. The amount of storage consumed by the 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 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** 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** 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^{ 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.

- **mipmapPrecisionBits** is the number of bits of division that the LOD calculation for mipmap fetching get snapped to when determining the contribution from each mip level to the mip filtered results. \( mipmapPrecisionBits \) is the actual number of divisions.

- **maxDrawIndexedIndexValue** is the maximum index value that can be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of 0xFFFFFFFF. See fullDrawIndexUint32.

- **maxDrawIndirectCount** is the maximum draw count that is supported for indirect draw calls. See multiDrawIndirect.

- **maxSamplerLodBias** is the maximum absolute sampler LOD bias. The sum of the \( mipLodBias \) member of the VkSamplerCreateInfo structure and the Bias operand of image sampling operations in shader modules (or 0 if no Bias operand is provided to an image sampling operation) are clamped to the range \([-maxSamplerLodBias,+maxSamplerLodBias]\). See [samplers-mipLodBias].

- **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 maxAnisotropy member of the VkSamplerCreateInfo structure and this limit. See [samplers-maxAnisotropy].

- **maxViewports** is the maximum number of active viewports. The viewportCount member of the VkPipelineViewportStateCreateInfo structure that is provided at pipeline creation must be less than or equal to this limit.

- **maxViewportDimensions[2]** are the maximum viewport dimensions in the X (width) and Y (height) dimensions, respectively. The maximum viewport dimensions must be greater than or equal to the largest image which can be created and used as a framebuffer attachment. See Controlling the Viewport.

- **viewportBoundsRange[2]** is the [minimum, maximum] range that the corners of a viewport must be contained in. This range must be at least \([-2 \times size, 2 \times size - 1]\), where \( size = \max(maxViewportDimensions[0], maxViewportDimensions[1])\). See Controlling the Viewport.

\[
\text{Note}
\]

The intent of the viewportBoundsRange limit is to allow a maximum sized viewport to be arbitrarily shifted relative to the output target as long as at least some portion intersects. This would give a bounds limit of \([-size + 1, 2 \times size - 1]\) which would allow all possible non-empty-set intersections of the output target and the viewport. Since these numbers are typically powers of two, picking the signed number range using the smallest possible number of bits ends up with the specified range.

- **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.

- **minMemoryMapAlignment** is the minimum required alignment, in bytes, of host visible memory allocations within the host address space. When mapping a memory allocation with vkMapMemory, subtracting offset bytes from the returned pointer will always produce an 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. 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 minimum negative offset value for the offset operand of the InterpolateAtOffset extended instruction.

• **maxInterpolationOffset** is the maximum positive offset value for the offset operand of the InterpolateAtOffset extended instruction.

• **subPixelInterpolationOffsetBits** is the number of subpixel 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 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 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 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 structure must be less than or equal to this limit.

- sampledImageColorSampleCounts is a bitmask 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 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 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 of VkSampleCountFlagBits indicating the sample 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 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** is the range \([\text{minimum}, \text{maximum}]\) of supported sizes for points. Values written to variables decorated with the `PointSize` built-in decoration are clamped to this range.

- **lineWidthRange** is the range \([\text{minimum}, \text{maximum}]\) 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 `vkCmdCopyBufferToImage` 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 `vkCmdCopyBufferToImage` 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.

1 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:
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.

typedef VkFlags VkSampleCountFlags;

VkSampleCountFlags is a bitmask type for setting a mask of zero or more VkSampleCountFlagBits.

### 31.1. 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 supported. If an implementation supports a feature, the limits reported are the same whether or not the feature is enabled.

<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>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------</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>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>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>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>Type</td>
<td>Limit</td>
<td>Feature</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------</td>
<td>--------------------------------</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>fullDrawIndexUint32</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>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>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>maxFramebufferWidth</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxFramebufferHeight</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxFramebufferLayers</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>framebufferColorSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>framebufferDepthSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>framebufferStencilSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>framebufferNoAttachmentsSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxColorAttachments</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>sampledImageColorSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>sampledImageIntegerSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>sampledImageDepthSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>sampledImageStencilSampleCounts</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkSampleCountFlags</code></td>
<td><code>storageImageSampleCounts</code></td>
<td>shaderStorageImageMultisample</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxSampleMaskWords</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>timestampComputeAndGraphics</code></td>
<td>-</td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>timestampPeriod</code></td>
<td>-</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxClipDistances</code></td>
<td>shaderClipDistance</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxCullDistances</code></td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>maxCombinedClipAndCullDistances</code></td>
<td>shaderCullDistance</td>
</tr>
<tr>
<td><code>uint32_t</code></td>
<td><code>discreteQueuePriorities</code></td>
<td>-</td>
</tr>
<tr>
<td><code>2 × float</code></td>
<td><code>pointSizeRange</code></td>
<td>largePoints</td>
</tr>
<tr>
<td><code>2 × float</code></td>
<td><code>lineWidthRange</code></td>
<td>wideLines</td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>pointSizeGranularity</code></td>
<td>largePoints</td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>lineWidthGranularity</code></td>
<td>wideLines</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>strictLines</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkBool32</code></td>
<td><code>standardSampleLocations</code></td>
<td>-</td>
</tr>
<tr>
<td><code>VkDeviceSize</code></td>
<td><code>optimalBufferCopyOffsetAlignment</code></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>VkDeviceSize</td>
<td>optimalBufferCopyRowPitchAlignment</td>
<td>-</td>
</tr>
<tr>
<td>VkDeviceSize</td>
<td>nonCoherentAtomSize</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 31. Required Limits**

<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupport ed 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>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>maxDescriptorSetSampledImages</td>
<td>-</td>
<td>96 ⁸</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetStorageImages</td>
<td>-</td>
<td>24 ⁸</td>
<td>min, n × PerStage</td>
</tr>
<tr>
<td>maxDescriptorSetInputAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputAttributes</td>
<td>-</td>
<td>16</td>
<td>min</td>
</tr>
<tr>
<td>maxVertexInputBindings</td>
<td>-</td>
<td>16</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>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>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>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)</td>
<td>min</td>
</tr>
<tr>
<td>viewportBoundsRange</td>
<td>-</td>
<td>(-8192,8191)</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>-0.5</td>
<td>max</td>
</tr>
<tr>
<td>maxInterpolationOffset</td>
<td>0.0</td>
<td>0.5 - (1 ULP)</td>
<td>min</td>
</tr>
<tr>
<td>subPixelInterpolationOffsetBits</td>
<td>0</td>
<td>4</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></td>
<td>VK_SAMPLE_COUNT_4_BIT)</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></td>
<td>VK_SAMPLE_COUNT_4_BIT)</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>framebufferStencilSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>framebufferNoAttachmentsSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>maxColorAttachments</td>
<td>-</td>
<td>4</td>
<td>min</td>
</tr>
<tr>
<td>sampledImageColorSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>sampledImageIntegerSampleCounts</td>
<td>-</td>
<td>VK_SAMPLE_COUNT_1_BIT</td>
<td>min</td>
</tr>
<tr>
<td>sampledImageDepthSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>sampledImageStencilSampleCounts</td>
<td>-</td>
<td>(VK_SAMPLE_COUNT_1_BIT</td>
<td>VK_SAMPLE_COUNT_4_BIT)</td>
</tr>
<tr>
<td>storageImageSampleCounts</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>maxSampleMaskWords</td>
<td>-</td>
<td>1</td>
<td>min</td>
</tr>
<tr>
<td>timestampComputeAndGraphics</td>
<td>-</td>
<td>-</td>
<td>implementatio n dependent</td>
</tr>
<tr>
<td>timestampPeriod</td>
<td>-</td>
<td>-</td>
<td>duration</td>
</tr>
<tr>
<td>maxClipDistances</td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>maxCullDistances</td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>maxCombinedClipAndCullDistances</td>
<td>0</td>
<td>8</td>
<td>min</td>
</tr>
<tr>
<td>discreteQueuePriorities</td>
<td>-</td>
<td>2</td>
<td>min</td>
</tr>
<tr>
<td>pointSizeRange</td>
<td>(1.0,1.0)</td>
<td>(1.0,64.0 - ULP)⁶</td>
<td>(max,min)</td>
</tr>
<tr>
<td>lineWidthRange</td>
<td>(1.0,1.0)</td>
<td>(1.0,8.0 - ULP)⁷</td>
<td>(max,min)</td>
</tr>
<tr>
<td>pointSizeGranularity</td>
<td>0.0</td>
<td>1.0⁶</td>
<td>max, fixed point increment</td>
</tr>
</tbody>
</table>

¹ Limit Type: min (minimum), max (maximum), duration, implementation dependent.
<table>
<thead>
<tr>
<th>Limit</th>
<th>Unsupported Limit</th>
<th>Supported Limit</th>
<th>Limit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineWidthGranularity</td>
<td>0.0</td>
<td>1.0⁷</td>
<td>max, fixed point increment</td>
</tr>
<tr>
<td>strictLines</td>
<td>-</td>
<td>-</td>
<td>implementation dependent</td>
</tr>
<tr>
<td>standardSampleLocations</td>
<td>-</td>
<td>-</td>
<td>implementation dependent</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>
</tbody>
</table>

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.

4. See **viewportBoundsRange** for the **required** relationship to other limits.

5. 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).
Chapter 32. 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.

32.1. 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
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,
    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_SNORM = 38,
VK_FORMAT_R8G8B8A8_USCALED = 39,
VK_FORMAT_R8G8B8A8_SSCALED = 40,
VK_FORMAT_R8G8B8A8_UINT = 41,
VK_FORMAT_R8G8B8A8_SINT = 42,
VK_FORMAT_R8G8B8A8_SRGB = 43,
VK_FORMAT_B8G8R8A8_UNORM = 44,
VK_FORMAT_B8G8R8A8_SNORM = 45,
VK_FORMAT_B8G8R8A8_USCALED = 46,
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 = 52,
VK_FORMAT_A8B8G8R8_USCALED_PACK32 = 53,
VK_FORMAT_A8B8G8R8_SSCALED_PACK32 = 54,
VK_FORMAT_A8B8G8R8_UINT_PACK32 = 55,
VK_FORMAT_A8B8G8R8_SINT_PACK32 = 56,
VK_FORMAT_A8B8G8R8_SRGB_PACK32 = 57,
VK_FORMAT_A2R10G10B10_UNORM_PACK32 = 58,
VK_FORMAT_A2R10G10B10_SNORM_PACK32 = 59,
VK_FORMAT_A2R10G10B10_USCALED_PACK32 = 60,
VK_FORMAT_A2R10G10B10_SSCALED_PACK32 = 61,
VK_FORMAT_A2R10G10B10_UINT_PACK32 = 62,
VK_FORMAT_A2R10G10B10_SINT_PACK32 = 63,
VK_FORMAT_A2B10G10R10_UNORM_PACK32 = 64,
VK_FORMAT_A2B10G10R10_SNORM_PACK32 = 65,
VK_FORMAT_A2B10G10R10_USCALED_PACK32 = 66,
VK_FORMAT_A2B10G10R10_SSCALED_PACK32 = 67,
VK_FORMAT_A2B10G10R10_UINT_PACK32 = 68,
VK_FORMAT_A2B10G10R10_SINT_PACK32 = 69,
VK_FORMAT_R16_UNORM = 70,
VK_FORMAT_R16_SNORM = 71,
VK_FORMAT_R16_USCALED = 72,
VK_FORMAT_R16_SSCALED = 73,
VK_FORMAT_R16_UINT = 74,
VK_FORMAT_R16_SINT = 75,
VK_FORMAT_R16_SFLOAT = 76,
VK_FORMAT_R16G16_UNORM = 77,
VK_FORMAT_R16G16_SNORM = 78,
VK_FORMAT_R16G16_USCALED = 79,
VK_FORMAT_R16G16_SSCALED = 80,
VK_FORMAT_R16G16_UINT = 81,
VK_FORMAT_R16G16_SINT = 82,
VK_FORMAT_R16G16_SFLOAT = 83,
<table>
<thead>
<tr>
<th>VK FORMAT</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R16G16B16_UNORM</td>
<td>84</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16_SNORM</td>
<td>85</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16_USCALED</td>
<td>86</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16_SSCALED</td>
<td>87</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16_UINT</td>
<td>88</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16_SINT</td>
<td>89</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16_SFLOAT</td>
<td>90</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_UNORM</td>
<td>91</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SNORM</td>
<td>92</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_USCALED</td>
<td>93</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SSCALED</td>
<td>94</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_UINT</td>
<td>95</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SINT</td>
<td>96</td>
</tr>
<tr>
<td>VK_FORMAT_R16G16B16A16_SFLOAT</td>
<td>97</td>
</tr>
<tr>
<td>VK_FORMAT_R32_UINT</td>
<td>98</td>
</tr>
<tr>
<td>VK_FORMAT_R32_SINT</td>
<td>99</td>
</tr>
<tr>
<td>VK_FORMAT_R32_SFLOAT</td>
<td>100</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32_UINT</td>
<td>101</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32_SINT</td>
<td>102</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32_SFLOAT</td>
<td>103</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_UINT</td>
<td>104</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_SINT</td>
<td>105</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32_SFLOAT</td>
<td>106</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
<td>107</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_SINT</td>
<td>108</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_SFLOAT</td>
<td>109</td>
</tr>
<tr>
<td>VK_FORMAT_R64_UINT</td>
<td>110</td>
</tr>
<tr>
<td>VK_FORMAT_R64_SINT</td>
<td>111</td>
</tr>
<tr>
<td>VK_FORMAT_R64_SFLOAT</td>
<td>112</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64_UINT</td>
<td>113</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64_SINT</td>
<td>114</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64_SFLOAT</td>
<td>115</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64B64_UINT</td>
<td>116</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64B64_SINT</td>
<td>117</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64B64_SFLOAT</td>
<td>118</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64B64A64_UINT</td>
<td>119</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64B64A64_SINT</td>
<td>120</td>
</tr>
<tr>
<td>VK_FORMAT_R64G64B64A64_SFLOAT</td>
<td>121</td>
</tr>
<tr>
<td>VK_FORMAT_B10G11R11_UFLOAT_PACK32</td>
<td>122</td>
</tr>
<tr>
<td>VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</td>
<td>123</td>
</tr>
<tr>
<td>VK_FORMAT_D16_UNORM</td>
<td>124</td>
</tr>
<tr>
<td>VK_FORMAT_X8_D24_UNORM_PACK32</td>
<td>125</td>
</tr>
<tr>
<td>VK_FORMAT_D32_SFLOAT</td>
<td>126</td>
</tr>
<tr>
<td>VK_FORMAT_S8_UINT</td>
<td>127</td>
</tr>
<tr>
<td>VK_FORMAT_D16_UNORM_S8_UINT</td>
<td>128</td>
</tr>
<tr>
<td>VK_FORMAT_D24_UNORM_S8_UINT</td>
<td>129</td>
</tr>
<tr>
<td>VK_FORMAT_D32_SFLOAT_S8_UINT</td>
<td>130</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
<td>131</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
<td>132</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
<td>133</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
<td>134</td>
</tr>
</tbody>
</table>
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_BC6H_UFLOAT_BLOCK = 143,
VK_FORMAT_BC6H_SFLOAT_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_6x5_UNORM_BLOCK = 161,
VK_FORMAT_ASTC_6x5_SRGB_BLOCK = 162,
VK_FORMAT_ASTC_6x6_UNORM_BLOCK = 163,
VK_FORMAT_ASTC_6x6_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_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_R5G6B5_UNORM_PACK16 specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit B component in bits 0..4.

VK_FORMAT_B5G6R5_UNORM_PACK16 specifies a three-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit R component in bits 0..4.

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 1, and an 8-bit B component stored with sRGB nonlinear encoding 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.
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 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_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 B 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 B 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 the first 64 bits encoding alpha values followed by 64 bits encoding RGB values 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 RGBA 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 RGBA texel data with sRGB nonlinear encoding, and provides 1 bit of alpha.

- **VK_FORMAT_ETC2_R8G8B8AB_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 with the first 64 bits encoding alpha values followed by 64 bits encoding RGB values.

- **VK_FORMAT_ETC2_R8G8B8AB_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_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_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_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_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_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_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_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 with sRGB nonlinear encoding applied to the RGB components.

• **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_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_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_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_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_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.

### 32.1.1. 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
  - 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

- **Packed into 32-bit data types:**
  - 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_A8B8G8R8_SRGB_PACK32
  - 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
32.1.2. 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:

```
VK_FORMAT_{component-format|compression-scheme}_{numeric-format}
```

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 32. 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>

The suffix _PACKnn indicates that the format is 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 33. 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>

32.1.3. 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 34. Byte mappings for non-packed/compressed color formats*

<table>
<thead>
<tr>
<th>Byte</th>
<th>R</th>
<th>VK_FORMAT_R8_::*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>VK_FORMAT_R8G8_::*</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>VK_FORMAT_R8G8B8_::*</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>VK_FORMAT_B8G8R8_::*</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>VK_FORMAT_R8G8B8A8_::*</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>VK_FORMAT_B8G8R8A8_::*</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>VK_FORMAT_R16_::*</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>VK_FORMAT_R16G16_::*</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>VK_FORMAT_R16G16B16_::*</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>VK_FORMAT_R16G16B16A16_::*</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>VK_FORMAT_R32_::*</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>VK_FORMAT_R32G32_::*</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>VK_FORMAT_R32G32B32_::*</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>VK_FORMAT_R32G32B32A32_::*</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>VK_FORMAT_R64_::*</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>VK_FORMAT_R64G64_::*</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>VK_FORMAT_R64G64B64_::* as VK_FORMAT_R64G64_::* but with B in bytes 16-23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VK_FORMAT_R64G64B64A64_::* as VK_FORMAT_R64G64B64_::* but with A in bytes 24-31</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 35. Bit mappings for packed 8-bit formats*

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[VK\_FORMAT\_R4G4\_UNORM\_PACK8\]
### Table 36. Bit mappings for packed 16-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R4G4B4A4_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**VK_FORMAT_B4G4R4A4_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R5G6B5_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**VK_FORMAT_B5G6R5_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**VK_FORMAT_R5G5B5A1_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**VK_FORMAT_B5G5R5A1_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A1R5G5B5_UNORM_PACK16**

<table>
<thead>
<tr>
<th>Bit</th>
<th>A</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 37. Bit mappings for packed 32-bit formats

<table>
<thead>
<tr>
<th>Bit</th>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A8B8G8R8_*_PACK32**

<table>
<thead>
<tr>
<th>Bit</th>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A2R10G10B10_*_PACK32**

<table>
<thead>
<tr>
<th>Bit</th>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**VK_FORMAT_A2B10G10R10_*_PACK32**

<table>
<thead>
<tr>
<th>Bit</th>
<th>A</th>
<th>B</th>
<th>G</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

664 | Chapter 32. Formats
32.1.4. 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).

32.1.5. 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 38. 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, VK_FORMAT_R8_UNORM, VK_FORMAT_R8_SNORM, VK_FORMAT_R8_USCALED, VK_FORMAT_R8_SSCALED, VK_FORMAT_R8_UINT, VK_FORMAT_R8_SINT, VK_FORMAT_R8_SRGB</td>
</tr>
<tr>
<td>Block size 1 byte 1 texel/block</td>
<td></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_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</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_B8G8R8A8_SRGB, 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_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_R16G16_UNORM, VK_FORMAT_R16G16_SNORM, VK_FORMAT_R16G16_USCALED, VK_FORMAT_R16G16_SSCALED, VK_FORMAT_R16G16_UINT, VK_FORMAT_R16G16_SINT, VK_FORMAT_R32_UINT, VK_FORMAT_R32_SINT, VK_FORMAT_R32_SFLOAT, VK_FORMAT_B10G11R11_UFLOAT_PACK32, VK_FORMAT_E5B9G9R9_UFLOAT_PACK32</td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Block size 6 bytes 1 texel/block</td>
<td></td>
</tr>
<tr>
<td>Block size 8 bytes 1 texel/block</td>
<td></td>
</tr>
<tr>
<td><strong>96-bit</strong></td>
<td>VK_FORMAT_R32G32B32_UINT, VK_FORMAT_R32G32B32_SINT, VK_FORMAT_R32G32B32_SFLOAT</td>
</tr>
<tr>
<td>Block size 12 bytes 1 texel/block</td>
<td></td>
</tr>
<tr>
<td><strong>128-bit</strong></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>Block size 16 bytes 1 texel/block</td>
<td></td>
</tr>
<tr>
<td><strong>192-bit</strong></td>
<td>VK_FORMAT_R64G64B64_UINT, VK_FORMAT_R64G64B64_SINT, VK_FORMAT_R64G64B64_SFLOAT</td>
</tr>
<tr>
<td>Block size 24 bytes 1 texel/block</td>
<td></td>
</tr>
<tr>
<td><strong>256-bit</strong></td>
<td>VK_FORMAT_R64G64B64A64_UINT, VK_FORMAT_R64G64B64A64_SINT, VK_FORMAT_R64G64B64A64_SFLOAT</td>
</tr>
<tr>
<td>Block size 32 bytes 1 texel/block</td>
<td></td>
</tr>
<tr>
<td><strong>BC1_RGB (64 bit)</strong></td>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK</td>
</tr>
<tr>
<td>Block size 8 bytes 16 texels/block</td>
<td></td>
</tr>
<tr>
<td><strong>BC1_RGBA (64 bit)</strong></td>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK</td>
</tr>
<tr>
<td>Block size 8 bytes 16 texels/block</td>
<td></td>
</tr>
<tr>
<td>Class, Texel Block Size, # Texels/Block</td>
<td>Formats</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------</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>BC5 (128 bit) Block size 16 bytes 16 texels/block</td>
<td>VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK</td>
</tr>
<tr>
<td>BC6H (128 bit) Block size 16 bytes 16 texels/block</td>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
</tr>
<tr>
<td>BC7 (128 bit) Block size 16 bytes 16 texels/block</td>
<td>VK_FORMAT_BC7_UNORM_BLOCK, VK_FORMAT_BC7_SRGB_BLOCK</td>
</tr>
<tr>
<td>ETC2_RGB (64 bit) Block size 8 bytes 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) Block size 8 bytes 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) Block size 8 bytes 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) Block size 8 bytes 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) Block size 16 bytes 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) Block size 16 bytes 16 texels/block</td>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_5x4 (128 bit) Block size 16 bytes 20 texels/block</td>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_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_5x5 (128 bit) Block size 16 bytes 25 texels/block</td>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_6x5 (128 bit) Block size 16 bytes 30 texels/block</td>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
</tr>
<tr>
<td>ASTC_6x6 (128 bit) Block size 16 bytes 36 texels/block</td>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK</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_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_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_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_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_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_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_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_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_SRGB_BLOCK</td>
</tr>
<tr>
<td>D16 (16 bit) Block size 2 bytes 1 texel/block</td>
<td>VK_FORMAT_D16_UNORM</td>
</tr>
</tbody>
</table>
### 32.2. Format Properties

To query supported format features which are properties of the physical device, call:

```c
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)

- `physicalDevice` must be a valid `VkPhysicalDevice` handle
- `format` must be a valid `VkFormat` value
- `pFormatProperties` must be a valid pointer to a `VkFormatProperties` structure

The `VkFormatProperties` structure is defined as:
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.

- **bufferFeatures** is a bitmask of VkFormatFeatureFlagBits specifying features supported by buffers.

**Note**

If no format feature flags are supported, then the only possible use would be image transfers - which alone are not useful. As such, 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.

Bits which **can** be set in the VkFormatProperties features `linearTilingFeatures`, `optimalTilingFeatures`, and `bufferFeatures` are:

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,
} VkFormatFeatureFlagBits;

The following bits **may** be set in `linearTilingFeatures` and `optimalTilingFeatures`, specifying that the features are supported by images or image views 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 images.

• **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 `vkCmdBlitImage` command.

• **VK_FORMAT_FEATURE_BLIT_DST_BIT** specifies that an image can be used as dstImage for the `vkCmdBlitImage` command.

• **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 `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.

The following bits may be set in `bufferFeatures`, specifying that the features are supported by buffers or buffer views created with the queried `vkGetPhysicalDeviceProperties::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**).
typedef VkFlags VkFormatFeatureFlags;

VkFormatFeatureFlags is a bitmask type for setting a mask of zero or more VkFormatFeatureFlagBits.

32.3. 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.

**Table 39. Key for format feature tables**

| ✓  | This feature **must** be supported on the named format |
| †  | This feature **must** be supported on at least some of the named formats, with more information in the table where the symbol appears |
| ‡  | This feature **must** be supported with some caveats or preconditions, with more information in the table where the symbol appears |

**Table 40. Feature bits in optimalTilingFeatures**

- 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

**Table 41. Feature bits in bufferFeatures**

- VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT
- VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT

674 | Chapter 32. Formats
<table>
<thead>
<tr>
<th>Feature Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</td>
</tr>
</tbody>
</table>
Table 42. Mandatory format support: sub-byte 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_UNDEFINED</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_R4G4_UNORM_PACK8</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_R4G4B4A4_UNORM_PACK16</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_B4G4R4A4_UNORM_PACK16</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_R5G6B5_UNORM_PACK16</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_B5G5R5A1_UNORM_PACK16</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_A1R5G5B5_UNORM_PACK16</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 43. Mandatory format support: 1-3 byte-sized channels

<table>
<thead>
<tr>
<th>Format</th>
<th>VK_FORMAT_R8_UNORM</th>
<th>VK_FORMAT_R8_SNORM</th>
<th>VK_FORMAT_R8_USCALED</th>
<th>VK_FORMAT_R8_SSCALED</th>
<th>VK_FORMAT_R8_UINT</th>
<th>VK_FORMAT_R8_SINT</th>
<th>VK_FORMAT_R8_SRGB</th>
<th>VK_FORMAT_R8G8B8_UNORM</th>
<th>VK_FORMAT_R8G8B8_SNORM</th>
<th>VK_FORMAT_R8G8B8_USCALED</th>
<th>VK_FORMAT_R8G8B8_SSCALED</th>
<th>VK_FORMAT_R8G8B8_UINT</th>
<th>VK_FORMAT_R8G8B8_SINT</th>
<th>VK_FORMAT_R8G8B8_SRGB</th>
<th>VK_FORMAT_R8G8B8B8_UNORM</th>
<th>VK_FORMAT_R8G8B8B8_SNORM</th>
<th>VK_FORMAT_R8G8B8B8_USCALED</th>
<th>VK_FORMAT_R8G8B8B8_SSCALED</th>
<th>VK_FORMAT_R8G8B8B8_UINT</th>
<th>VK_FORMAT_R8G8B8B8_SINT</th>
<th>VK_FORMAT_R8G8B8B8_SRGB</th>
<th>VK_FORMAT_R8G8B8B8_SRGB</th>
<th>VK_FORMAT_B8G8R8_UNORM</th>
<th>VK_FORMAT_B8G8R8_SNORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_B8G8R8_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>
<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>--------------------------</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></td>
</tr>
<tr>
<td>VK_FORMAT_B8G8R8_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>
<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_B8G8R8_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>
<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_B8G8R8_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>
<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_B8G8R8_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>
<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>

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 44. 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>
<th>Format</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>
<td>VK_FORMAT_R8G8B8A8_SINT</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>
<td>VK_FORMAT_R8G8B8A8_SRGB</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>
<td>VK_FORMAT_R8G8B8A8_SRGB</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>
<td>VK_FORMAT_R8G8B8A8_SRGB</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>
<td>VK_FORMAT_R8G8B8A8_SRGB</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>
<td>VK_FORMAT_R8G8B8A8_SRGB</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>
<td>VK_FORMAT_R8G8B8A8_SRGB</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>
<td>VK_FORMAT_B8G8R8A8_SRCB</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>
<td>VK_FORMAT_B8G8R8A8_SRCB</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>
<td>VK_FORMAT_B8G8R8A8_SRCB</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>
<td>VK_FORMAT_B8G8R8A8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</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>
<td>VK_FORMAT_A8B8G8R8_SRCB</td>
</tr>
</tbody>
</table>

Chapter 32. Formats | 679
Table 45. Mandatory format support: 10- and 12-bit channels

<table>
<thead>
<tr>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</th>
<th>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</td>
<td>VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</td>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT</td>
<td>VK_FORMAT_FEATURE_BLIT_DST_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</td>
<td>VK_FORMAT_FEATURE_BLIT_SRC_BIT</td>
</tr>
<tr>
<td>VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT</td>
<td></td>
</tr>
</tbody>
</table>

Format

<table>
<thead>
<tr>
<th>Format</th>
<th>O</th>
<th>O</th>
<th>V</th>
<th>O</th>
<th>O</th>
<th>O</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_A2R10G10B10_UNORM_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2R10G10B10_SNORM_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2R10G10B10_USCALED_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2R10G10B10_SSCALED_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2R10G10B10_UINT_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2R10G10B10_SINT_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2B10G10R10_UNORM_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2B10G10R10_SNORM_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2B10G10R10_USCALED_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2B10G10R10_SSCALED_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2B10G10R10_UINT_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_A2B10G10R10_SINT_PACK32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
<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>VK_FORMAT_R16G16B16A16_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>
<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_R16G16B16A16_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>
<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_R16G16B16A16_SFLOAT</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>
<tr>
<td>VK_FORMAT_R16G16B16A16_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>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Format Features</td>
<td>Support</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>
<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>
<tr>
<td>VK_FORMAT_R16G16B16A16_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>
<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_R16G16B16A16_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>
<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_R16G16B16A16_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>
<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_R16G16B16A16_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>
<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_R16G16B16A16_SFLOAT</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>

Format features marked with ‡ must be supported for optimal Tiling Features if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.
Table 47. 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>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_R32G32B32A32_UINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_SINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VK_FORMAT_R32G32B32A32_SFLOAT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Table 48. Mandatory format support: 64-bit/uneven channels

<table>
<thead>
<tr>
<th>Format features marked with ‡ must be supported for optimalTilingFeatures if the VkPhysicalDevice supports the shaderStorageImageExtendedFormats feature.</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>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</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_FEATURE_STORAGE_TEXEL_BUFFER_BIT</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_FEATURE_UNIFORM_TEXEL_BUFFER_BIT</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_FEATURE_VERTEX_BUFFER_BIT</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_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT</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_FEATURE_COLOR_ATTACHMENT_BLEND_BIT</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_FEATURE_BLIT_DST_BIT</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_FEATURE_COLOR_ATTACHMENT_BIT</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_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</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_FEATURE_STORAGE_IMAGE_ATOMIC_BIT</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_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT</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_FEATURE_BLIT_SRC_BIT</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_FEATURE_SAMPLED_IMAGE_ATOMIC_BIT</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_FEATURE_COLOR_ATTACHMENT_BIT</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_FEATURE_SAMPLED_IMAGE_ATOMIC_BIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</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_R64_UINT</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_R64_SINT</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_R64_SFLOAT</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_R64G64_UINT</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_R64G64_SINT</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_R64G64_SFLOAT</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_R64G64B64_UINT</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_R64G64B64_SINT</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_R64G64B64A64_UINT</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_R64G64B64A64_SINT</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_R64G64B64A64_SFLOAT</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_B10G11R11_UFLOAT_PACK32</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_E5B9G9R9_UFLOAT_PACK32</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 49. Mandatory format support: depth/stencil with VkImageType VK_IMAGE_TYPE_2D

<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_D16_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_X8_D24_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_D32_SFLOAT</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_S8_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_D16_UNORM_S8_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_D24_UNORM_S8_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_D32_SFLOAT_S8_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>
</tbody>
</table>

The **feature** must be supported for at least one of **VK_FORMAT_X8_D24_UNORM_PACK32** and **VK_FORMAT_D32_SFLOAT**, and **must** be supported for at least one of **VK_FORMAT_D24_UNORM_S8_UINT** and **VK_FORMAT_D32_SFLOAT_S8_UINT**.
Table 50. Mandatory format support: BC compressed formats with VkImageType VK_IMAGE_TYPE_2D and VK_IMAGE_TYPE_3D

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_BC1_RGB_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC1_RGBA_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC2_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC3_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC4_SNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC5_SNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_UFLOAT_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC6H_SFLOAT_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_UNORM_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_RGBA_SRGB_BLOCK</td>
</tr>
<tr>
<td>VK_FORMAT_BC7_RGB_SRGB_BLOCK</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 51. Mandatory format support: ETC2 and EAC compressed formats with `VkImageType` `VK_IMAGE_TYPE_2D`

<table>
<thead>
<tr>
<th>Format</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_EAC_R11_UNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_EAC_R11_SNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_EAC_R11G11_UNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></td>
</tr>
<tr>
<td><code>VK_FORMAT_EAC_R11G11_SNORM_BLOCK</code></td>
<td><code>VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT</code></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: BC compressed formats with `VkImageType` `VK_IMAGE_TYPE_2D` and `VK_IMAGE_TYPE_3D`, or Mandatory format support: ASTC LDR compressed formats with `VkImageType` `VK_IMAGE_TYPE_2D`. 

---

**Chapter 32. Formats | 687**
Table 52. Mandatory format support: ASTC LDR compressed formats with VkImageType VK_IMAGE_TYPE_2D

<table>
<thead>
<tr>
<th>Format</th>
<th>bit features</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_FORMAT_ASTC_4x4_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>† † †</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
<td>†</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK</td>
<td>†</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_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**: 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**.
Chapter 33. Additional Capabilities

This chapter describes additional capabilities beyond the minimum capabilities described in the (Limits and Formats chapters, including:

- Additional Image Capabilities

33.1. 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
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 (Implicit)

- physicalDevice must be a valid VkPhysicalDevice handle
- format must be a valid VkFormat value
- type must be a valid VkImageType value
- tiling must be a valid VkImageTiling value
- usage must be a valid combination of VkImageUsageFlagBits values
- usage must not be 0
- 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
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

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

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.

33.1.1. 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
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` includes 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`.

### 33.1.2. Allowed Extent Values Based On Image Type

Implementations may support extent values larger than the required minimum/maximum values for certain types of images 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`:

- `maxExtent.width ≥ VkPhysicalDeviceLimits.maxImageDimension1D`
- `maxExtent.height = 1`
- `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_2D` when `flags` does not contain `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

- `maxExtent.width ≥ VkPhysicalDeviceLimits.maxImageDimension2D`
• `maxExtent.height ≥ VkPhysicalDeviceLimits.maxImageDimension2D`
• `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_2D` when `flags` contains `VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT`:

• `maxExtent.width ≥ VkPhysicalDeviceLimits.maxImageDimensionCube`
• `maxExtent.height ≥ VkPhysicalDeviceLimits.maxImageDimensionCube`
• `maxExtent.depth = 1`

For `VK_IMAGE_TYPE_3D`:

• `maxExtent.width ≥ VkPhysicalDeviceLimits.maxImageDimension3D`
• `maxExtent.height ≥ VkPhysicalDeviceLimits.maxImageDimension3D`
• `maxExtent.depth ≥ VkPhysicalDeviceLimits.maxImageDimension3D`
Chapter 34. 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
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_EVENT = 11,
    VK_OBJECT_TYPE_QUERY_POOL = 12,
    VK_OBJECT_TYPE_BUFFER_VIEW = 13,
    VK_OBJECT_TYPE_IMAGE_VIEW = 14,
    VK_OBJECT_TYPE_SHADER_MODULE = 15,
    VK_OBJECT_TYPE_PIPELINE_CACHE = 16,
    VK_OBJECT_TYPE_PIPELINE_LAYOUT = 17,
    VK_OBJECT_TYPE_RENDER_PASS = 18,
    VK_OBJECT_TYPE_PIPELINE = 19,
    VK_OBJECT_TYPE_DESCRIPTOR_SET_LAYOUT = 20,
    VK_OBJECT_TYPE_SAMPLER = 21,
    VK_OBJECT_TYPE_DESCRIPTOR_POOL = 22,
    VK_OBJECT_TYPE_DESCRIPTOR_SET = 23,
    VK_OBJECT_TYPE_FRAMEBUFFER = 24,
    VK_OBJECT_TYPE_COMMAND_POOL = 25,
} VkObjectType;
```

<table>
<thead>
<tr>
<th><code>VkObjectType</code></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>
</tbody>
</table>

*Table 53. VkObjectType and Vulkan Handle Relationship*
<table>
<thead>
<tr>
<th>VkObjectType</th>
<th>Vulkan Handle Type</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

If this Specification was generated with any such extensions included, they will be described in the remainder of this chapter.
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 1.0 implementation must support the 1.0 version of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL.

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 SPIR-V capabilities listed below must be supported if the corresponding feature or extension is enabled, or if no features or extensions are listed for that capability. Extensions are only listed when there is not also a feature bit associated with that capability.

Table 54. List of SPIR-V Capabilities and enabling features or extensions

<table>
<thead>
<tr>
<th>SPIR-V OpCapability</th>
<th>Vulkan feature or extension name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td></td>
</tr>
<tr>
<td>Shader</td>
<td></td>
</tr>
<tr>
<td>InputAttachment</td>
<td></td>
</tr>
<tr>
<td>Sampled1D</td>
<td></td>
</tr>
<tr>
<td>Image1D</td>
<td></td>
</tr>
<tr>
<td>SampledBuffer</td>
<td></td>
</tr>
<tr>
<td>ImageBuffer</td>
<td></td>
</tr>
<tr>
<td>ImageQuery</td>
<td></td>
</tr>
<tr>
<td>DerivativeControl</td>
<td></td>
</tr>
<tr>
<td>Geometry</td>
<td>geometryShader</td>
</tr>
<tr>
<td>Tessellation</td>
<td>tessellationShader</td>
</tr>
<tr>
<td>Float64</td>
<td>shaderFloat64</td>
</tr>
<tr>
<td>Int64</td>
<td>shaderInt64</td>
</tr>
<tr>
<td>Int16</td>
<td>shaderInt16</td>
</tr>
<tr>
<td>TessellationPointSize</td>
<td>shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>GeometryPointSize</td>
<td>shaderTessellationAndGeometryPointSize</td>
</tr>
<tr>
<td>ImageGatherExtended</td>
<td>shaderImageGatherExtended</td>
</tr>
<tr>
<td>StorageImageMultisample</td>
<td>shaderStorageImageMultisample</td>
</tr>
</tbody>
</table>
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.

**Validation Rules within a Module**

A SPIR-V module passed to `vkCreateShaderModule` **must** conform to the following rules:

- Every entry point **must** have no return value and accept no arguments.
- Recursion: The static function-call graph for an entry point **must** not contain cycles.
- The **Logical** addressing model **must** be selected.
- **Scope** for execution **must** be limited to:
  - **Workgroup**
    - The **Workgroup** scope **must** only be used in the tessellation control, and compute execution models.
  - **Subgroup**
- **Scope** for memory **must** be limited to:
  - **Device**
Device scope only extends to the queue family, not the whole device.

- **Workgroup**
  - The **WorkGroup** scope must only be used in the compute execution model(s).

- **Invocation**
  - **Storage Class must** be limited to:
    - UniformConstant
    - Input
    - Uniform
    - Output
    - Workgroup
      - The **Workgroup** storage class must only be used in the compute execution model(s).
      - Private
      - Function
      - PushConstant
      - Image
  - **Memory semantics must** obey the following rules:
    - **Acquire must** not be used with OpAtomicStore.
    - **Release must** not be used with OpAtomicLoad.
    - **AcquireRelease must** not be used with OpAtomicStore or OpAtomicLoad.
    - Sequentially consistent atomics and barriers are not supported and **SequentiallyConsistent** is treated as **AcquireRelease**. **SequentiallyConsistent should** not be used.
    - OpMemoryBarrier must use one of Acquire, Release, AcquireRelease, or **SequentiallyConsistent** and must include at least one storage class.
    - If the semantics for OpControlBarrier includes one of Acquire, Release, AcquireRelease, or **SequentiallyConsistent**, then it must include at least one storage class.
    - SubgroupMemory, CrossWorkgroupMemory, and AtomicCounterMemory are ignored.

- Any OpVariable with an **Initializer** operand must have one of the following as its **Storage Class** operand:
  - Output
  - Private
  - Function

- The **OriginLowerLeft** execution mode must not be used; fragment entry points must declare **OriginUpperLeft**.

- The **PixelCenterInteger** execution mode must not be used. Pixels are always centered at half-integer coordinates.

- Any variable in the **UniformConstant** storage class must be typed as either:
• OpTypeImage
• OpTypeSampler
• OpTypeSampledImage
• An array of one of these types.

• Images and Samplers
  • OpTypeImage must declare a scalar 32-bit float 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.)
  • If the Sampled Type of an OpTypeImage declaration does not match the numeric format of the corresponding resource in type, as shown in the SPIR-V Sampled Type column of the Interpretation of Numeric Format table, the values obtained by reading or sampling from the image are undefined.
  • If the signedness of any read or sample operation does not match the signedness of the corresponding resource then the values obtained are undefined.
  • OpTypeImage must have a “Sampled” operand of 1 (sampled image) or 2 (storage image).
  • 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.
  • 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 an OpImageTexelPointer is used in an atomic operation, the image type of the image parameter to OpImageTexelPointer must have an image format of R32i or R32ui.
  • OpImageQuerySizeLod, 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.
  • The “Depth” operand of OpTypeImage is ignored.
  • 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.

• The “Component” operand of OpImageGather, and OpImageSparseGather must be the <id> of a constant instruction.

• Structure types must not contain opaque types.

• Decorations
  • Any BuiltIn decoration not listed in Built-In Variables must not be used.
• Any `BuiltIn` decoration that corresponds only to Vulkan features or extensions that have not been enabled **must** not be used.

• The `GLSLShared` 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 `Patch` decoration **must** not be used on variables in the interface of a vertex, geometry, or fragment shader stage's entry point.

• `DescriptorSet` and `Binding` decorations **must** obey the constraints on storage class, type, and descriptor type described in `DescriptorSet and Binding Assignment`.

• 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.

• `OpTypeRuntimeArray` **must** only be used for:
  
  • the last member of an `OpTypeStruct` that is in the `Uniform` storage class decorated as `BufferBlock`.

• Specialization constants:
  
  • A type `T` that is an array sized with a specialization constant **can** be, or be contained in, the type of a Variable `V` only if:
    
    • `T` is the (top-level) type of `V`, or
    
    • `V` is declared in the `Function`, `Private`, or `Workgroup` storage classes, or
    
    • `V` is an interface variable with an additional level of arrayness, as described in interface matching, in which case `T` is allowed to be the element type of the (top-level) type of `V`.

• Linkage: See `Shader Interfaces` for additional linking and validation rules.

• Compute Shaders
  
  • For each compute shader entry point, either a `LocalSize` execution mode or an object decorated with the `WorkgroupSize` decoration **must** be specified.

• Atomic instructions **must** declare a scalar 32-bit integer type, for the value pointed to by `Pointer`.

• The `Pointer` operand of all atomic instructions **must** have a `Storage Class` limited to:
  
  • `Uniform`
  
  • `Workgroup`
  
  • `Image`

### 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.
• Any denormalized value input into a shader or potentially generated by any instruction in a shader (except those listed above) may be flushed to 0.
• The rounding mode cannot be set, and results will be correctly rounded, as described below.
• NaNs may not be generated. Instructions that operate on a NaN may not result in a NaN.

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 used is not defined but must obey 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, \(\text{ulp}(x) = \min_a |b - c|\). 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 \( E = \max(|x - F_{\text{min}}|, |x - F_{\text{max}}|)\).

For single precision (32 bit) instructions, precisions are required to be at least as follows, unless decorated with RelaxedPrecision:

### Table 55. Precision of core SPIR-V Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpFAdd</td>
<td>Correctly rounded.</td>
</tr>
<tr>
<td>OpFSub</td>
<td>Correctly rounded.</td>
</tr>
<tr>
<td>OpFOrdEqual, OpFUnordEqual</td>
<td>Correct result.</td>
</tr>
<tr>
<td>OpFOrdLessThan, OpFUnordLessThan</td>
<td>Correct result.</td>
</tr>
<tr>
<td>OpFOrdGreaterThan, OpFUnordGreaterThan</td>
<td>Correct result.</td>
</tr>
<tr>
<td>OpFOrdGreaterThanEqual, OpFUnordGreaterThanEqual</td>
<td>Correct result.</td>
</tr>
<tr>
<td>OpFDiv((x, y))</td>
<td>(2.5) ULP for (</td>
</tr>
</tbody>
</table>

conversions between types Correctly rounded.

### Table 56. Precision of GLSL.std.450 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{fma}())</td>
<td>Inherited from \text{OpFMul} followed by \text{OpFAdd}.</td>
</tr>
<tr>
<td>(\text{exp}(x), \text{exp2}(x))</td>
<td>(3 + 2 \times</td>
</tr>
<tr>
<td>(\text{log}(), \text{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>
</tr>
<tr>
<td>(\text{pow}(x, y))</td>
<td>Inherited from \text{exp2}(y \times \text{log2}(x)).</td>
</tr>
<tr>
<td>(\text{sqrt}())</td>
<td>Inherited from (1.0 / \text{inversesqrt}()).</td>
</tr>
<tr>
<td>(\text{inversesqrt}())</td>
<td>2 ULP.</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 \text{OpSRem} and \text{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.

Compatibility Between SPIR-V Image Formats And Vulkan Formats

Images which are read from or written to by shaders must have SPIR-V image formats compatible with the Vulkan image formats backing the image under the circumstances described for texture image validation. The compatible formats are:

Table 57. 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>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>Rg32f</td>
<td>VK_FORMAT_R32G32_SFLOAT</td>
</tr>
<tr>
<td>Rg16f</td>
<td>VK_FORMAT_R16G16_SFLOAT</td>
</tr>
<tr>
<td>R11fG11fB10f</td>
<td>VK_FORMAT_B10G11R11_UFLOAT_PACK32</td>
</tr>
<tr>
<td>R16f</td>
<td>VK_FORMAT_R16_SFLOAT</td>
</tr>
<tr>
<td>Rgba16</td>
<td>VK_FORMAT_R16G16B16A16_UNORM</td>
</tr>
<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_SINT</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>SPIR-V Image Format</td>
<td>Compatible Vulkan Format</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</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: 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 58. 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>Formats described in the “S3TC Compressed Texture Image Formats” chapter</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>Formats described in the “RGTC Compressed Texture Image Formats” chapter</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>Formats described in the “BPTC Compressed Texture Image Formats” chapter</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 59. 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 60. 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 x 4</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_4x4_SRGB_BLOCK</td>
<td>4 x 4</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_UNORM_BLOCK</td>
<td>5 x 4</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x4_SRGB_BLOCK</td>
<td>5 x 4</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_UNORM_BLOCK</td>
<td>5 x 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_5x5_SRGB_BLOCK</td>
<td>5 x 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_UNORM_BLOCK</td>
<td>6 x 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x5_SRGB_BLOCK</td>
<td>6 x 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_UNORM_BLOCK</td>
<td>6 x 6</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_6x6_SRGB_BLOCK</td>
<td>6 x 6</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_UNORM_BLOCK</td>
<td>8 x 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x5_SRGB_BLOCK</td>
<td>8 x 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_UNORM_BLOCK</td>
<td>8 x 6</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x6_SRGB_BLOCK</td>
<td>8 x 6</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_UNORM_BLOCK</td>
<td>8 x 8</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_8x8_SRGB_BLOCK</td>
<td>8 x 8</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_UNORM_BLOCK</td>
<td>10 x 5</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x5_SRGB_BLOCK</td>
<td>10 x 5</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_UNORM_BLOCK</td>
<td>10 x 6</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x6_SRGB_BLOCK</td>
<td>10 x 6</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_UNORM_BLOCK</td>
<td>10 x 8</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x8_SRGB_BLOCK</td>
<td>10 x 8</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_UNORM_BLOCK</td>
<td>10 x 10</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_10x10_SRGB_BLOCK</td>
<td>10 x 10</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_UNORM_BLOCK</td>
<td>12 x 10</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x10_SRGB_BLOCK</td>
<td>12 x 10</td>
<td>Yes</td>
<td>LDR</td>
</tr>
<tr>
<td>VK_FORMAT_ASTC_12x12_UNORM_BLOCK</td>
<td>12 x 12</td>
<td>No</td>
<td>LDR</td>
</tr>
<tr>
<td>VkFormat</td>
<td>Compressed 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>
</tbody>
</table>

The ASTC decode mode is decode_float16.
Appendix C: 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.
Appendix D: 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 Extensions
Appendix E: 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:

```
VKAPI_ATTR <return_type> VKAPI_CALL <command_name>(<command_parameters>);
```

Additionally, a Vulkan function pointer type declaration takes the form of:

```
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.

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.

```
// Version of this file
#define VK_HEADER_VERSION 140
```

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 VK_API_VERSION_*_* values.
// Complete version of this file
#define VK_HEADER_VERSION_COMPLETE VK_MAKE_VERSION(1, 2, VK_HEADER_VERSION)

VK_API_VERSION is now commented out of vulkan_core.h and cannot be used.

// 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

**Vulkan Handle Macros**

**VK_DEFINE_HANDLE** defines a *dispatchable handle* type.

```c
#define VK_DEFINE_HANDLE(object) typedef struct object##_T* object;
```

- *object* is the name of the resulting C type.

The only dispatchable handle types are those related to device and instance management, such as VkDevice.

**VK_DEFINE_NON_DISPATCHABLE_HANDLE** defines a *non-dispatchable handle* type.

```c
#if !defined(VK_DEFINE_NON_DISPATCHABLE_HANDLE)
#if defined(__LP64__) || defined(_WIN64) || (defined(__x86_64__) && !defined(__ILP32__)) || defined(_M_X64) || defined(__ia64) || defined(_M_IA64) || defined(__aarch64__) || defined(__powerpc64__)
    #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
```

- *object* is the name of the resulting C type.

Most Vulkan handle types, such as VkBuffer, are non-dispatchable.
Note

The `vulkan_core.h` header allows the `VK_DEFINE_NON_DISPATCHABLE_HANDLE` definition to be overridden by the application. If `VK_DEFINE_NON_DISPATCHABLE_HANDLE` is already defined when `vulkan_core.h` is compiled, the default definition is 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
#define VK_NULL_HANDLE 0
```

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>
<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>vulkan_android.h</td>
<td>None</td>
<td>VK_USE_PLATFORM_ANDROID_KHR</td>
</tr>
<tr>
<td>[VK_KHR_wayland_surface]</td>
<td>Wayland</td>
<td>vulkan_wayland.h</td>
<td>&lt;wayland-client.h&gt;</td>
<td>VK_USE_PLATFORM_WAYLAND_KHR</td>
</tr>
</tbody>
</table>
### Extension Name

<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Window System Name</th>
<th>Platform-specific Header</th>
<th>Required External Headers</th>
<th>Controlling Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>[VK_KHR_win32_surface], [VK_KHR_external_memory_win32], [VK_KHR_win32_keyed_mutex], [VK_KHR_external_semaphore_win32], [VK_KHR_external_fence_win32],</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_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>
</tbody>
</table>

#### 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 Extension Header Control (Informative)

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, `vulkan_beta.h`, allowing applications to decide whether or not to include them. The mechanism is similar to window system-specific headers: before including `vulkan_beta.h`, applications must include `vulkan_core.h`.

#### Note

Sometimes a provisional extension will include a subset of its interfaces in `vulkan_core.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 `vulkan_beta.h`.

As a convenience for applications, `vulkan.h` conditionally includes `vulkan_beta.h`. Applications can control inclusion of `vulkan_beta.h` by defining the macro `VK_ENABLE_BETA_EXTENSIONS` before including `vulkan.h`. 
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 F: 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.

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, depends 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.
Glossary

The terms defined in this section are used consistently throughout this 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.

Adjacent Vertex

A vertex in an adjacency primitive topology that is not part of a given primitive, but is accessible in geometry shaders.

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. Others 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.

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 unsignaled. Represented by a VkSemaphore object.

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.

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 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 Renderable Format**
A VkFormat where VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT is set in one of the following, depending on the image's tiling:

- VkImageFormatProperties::linearTilingFeatures
- VkImageFormatProperties::optimalTilingFeatures

**Color Sample Mask**
A bitfield associated with a fragment, with one bit for each sample in the color attachment(s). Samples are considered to be covered based on the result of the Coverage Reduction stage. Uncovered samples do not write to color attachments.

**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 miplevel size. See Image Miplevel Sizing.

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.

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.
Deprecated (feature)
A feature is deprecated if it is no longer recommended as the correct or best way to achieve its intended purpose.

Depth/Stencil Attachment
A subpass attachment point, or image view, that is the target of depth and/or stencil test operations and writes.

Depth/Stencil Format
A VkFormat that includes depth and/or stencil components.

Depth/Stencil Image (or ImageView)
A VkImage (or VkImageView) with a depth/stencil format.

Derivative Group
A set of fragment shader invocations that cooperate to compute derivatives, including implicit derivatives for sampled image operations.

Descriptor
Information about a resource or resource view written into a descriptor set that is used to access the resource or view from a shader.

Descriptor Binding
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.

Descriptor Pool
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.

Descriptor Set
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.

Descriptor Set Layout
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 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 vkCmdDraw, and vkCmdDrawIndexed.

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

Commands that provoke work using a graphics pipeline. Includes vkCmdDraw, vkCmdDrawIndexed, vkCmdDrawIndirect, and vkCmdDrawIndexedIndirect.

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 complete 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.

**Extension Scope**

The set of objects and commands that can be affected by an extension. Extensions are either device scope or instance scope.

**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 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 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.

Global Workgroup
A collection of local workgroups dispatched by a single dispatch command.

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 `vkMapMemory`.

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.

**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.

**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`, and `vkCmdDrawIndexedIndirect`.

**Indirect Commands**
Drawing or dispatching commands that source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`, and `vkCmdDispatchIndirect`.

**Indirect Drawing Commands**
*Drawing commands* that source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, and `vkCmdDrawIndexedIndirect`. 
Initial State (Command Buffer)
A command buffer that has not begun recording commands. See also Recorded State and Executable State.

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.

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.

Linear Resource
A resource is linear if it is one of the following:

- a VkBuffer
- a VkImage created with VK_IMAGE_TILING_LINEAR

A resource is non-linear if it is one of the following:

- a VkImage created with VK_IMAGE_TILING_OPTIMAL

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.

**Minimum Miplevel Size**

The smallest size that is permitted for a miplevel. For conventional images this is 1x1x1. 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.

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, 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.

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.

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 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.

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.

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 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.

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.

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.

**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 Code**
A stream of instructions used to describe the operation of a shader.

**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.

**Side Effect**
A store to memory or atomic operation on memory from a shader invocation.

**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.

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, 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.

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.

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 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.

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

Abbreviations and acronyms are sometimes used in the Specification and the API where they are considered clear and commonplace, and are defined here:

**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 G: 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 (version 1.0, 1.1)
- Mathias Schott, NVIDIA (versions 1.0, 1.1)
- Matt Netsch, Qualcomm Technologies, Inc. (version 1.1)
- Matthäus Chajdas, AMD (version 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 Pietrasik, 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 Sundareshan, 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.