The OpenCL™ C 2.0 Specification

Document Revision: 40

Version 2.2-8, Mon, 08 Oct 2018 16:51:38 +0000

Table of Contents

Acknowledgements

The OpenCL specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Chuck Rose, Adobe
Eric Berdahl, Adobe
Shivani Gupta, Adobe
Bill Licea Kane, AMD
Ed Buckingham, AMD
Jan Civlin, AMD
Laurent Morichetti, AMD
Mark Fowler, AMD
Marty Johnson, AMD
Michael Mantor, AMD
Norm Rubin, AMD
Ofer Rosenberg, AMD
Brian Sumner, AMD
Victor Odintsov, AMD
Aaftab Munshi, Apple
Abe Stephens, Apple
Alexandre Namaan, Apple
Anna Tikhonova, Apple
Chendi Zhang, Apple
Eric Bainville, Apple
David Hayward, Apple
Giridhar Murthy, Apple
Ian Ollmann, Apple
Inam Rahman, Apple
James Shearer, Apple
MonPing Wang, Apple
Tanya Lattner, Apple
Mikael Bourges-Sevenier, Aptina
Anton Lokhmotov, ARM
Dave Shreiner, ARM
Hedley Francis, ARM
Robert Elliott, ARM
Scott Moyers, ARM
Tom Olson, ARM
Anastasia Stulova, ARM
Christopher Thompson-Walsh, Broadcom
Holger Waechtler, Broadcom
Norman Rink, Broadcom
Andrew Richards, Codeplay
Maria Rovatsou, Codeplay
Alistair Donaldson, Codeplay
Alastair Murray, Codeplay

Stephen Frye, Electronic Arts
Eric Schenk, Electronic Arts
Daniel Laroche, Freescale
David Neto, Google
Robin Grosman, Huawei
Craig Davies, Huawei
Brian Horton, IBM
Brian Watt, IBM
Gordon Fossum, IBM
Greg Bellows, IBM
Joaquin Madruga, IBM
Mark Nutter, IBM
Mike Perks, IBM
Sean Wagner, IBM
Jon Parr, Imagination Technologies
Robert Quill, Imagination Technologies
James McCarthy, Imagination Technologie
Jon Leech, Independent
Aaron Kunze, Intel
Aaron Lefohn, Intel
Adam Lake, Intel
Alexey Bader, Intel
Allen Hux, Intel
Andrew Brownsword, Intel
Andrew Lauritzen, Intel
Bartosz Sochacki, Intel
Ben Ashbaugh, Intel
Brian Lewis, Intel
Geoff Berry, Intel
Hong Jiang, Intel
Jayanth Rao, Intel
Josh Fryman, Intel
Larry Seiler, Intel
Mike MacPherson, Intel
Murali Sundaresan, Intel
Paul Lalonde, Intel
Raun Krisch, Intel
Stephen Junkins, Intel
Tim Foley, Intel
Timothy Mattson, Intel
Yariv Aridor, Intel
Michael Kinsner, Intel
Kevin Stevens, Intel
Benjamin Bergen, Los Alamos National Laboratory
Roy Ju, Mediatek
Bor-Sung Liang, Mediatek
Rahul Agarwal, Mediatek
Michal Witaszek, Mobica

JenqKuen Lee, NTHU
Amit Rao, NVIDIA
Ashish Srivastava, NVIDIA
Bastiaan Aarts, NVIDIA
Chris Cameron, NVIDIA
Christopher Lamb, NVIDIA
Dibyapran Sanyal, NVIDIA
Guatam Chakrabarti, NVIDIA
Ian Buck, NVIDIA
Jaydeep Marathe, NVIDIA
Jian-Zhong Wang, NVIDIA
Karthik Raghavan Ravi, NVIDIA
Kedar Patil, NVIDIA
Manjunath Kudlur, NVIDIA
Mark Harris, NVIDIA
Michael Gold, NVIDIA
Neil Trevett, NVIDIA
Richard Johnson, NVIDIA
Sean Lee, NVIDIA
Tushar Kashalikar, NVIDIA
Vinod Grover, NVIDIA
Xiangyun Kong, NVIDIA
Yogesh Kini, NVIDIA
Yuan Lin, NVIDIA
Mayuresh Pise, NVIDIA
Allan Tzeng, QUALCOMM
Alex Bourd, QUALCOMM
Anirudh Acharya, QUALCOMM
Andrew Gruber, QUALCOMM
Andrzej Mamona, QUALCOMM
Benedict Gaster, QUALCOMM
Bill Torzewski, QUALCOMM
Bob Rychlik, QUALCOMM
Chihong Zhang, QUALCOMM
Chris Mei, QUALCOMM
Colin Sharp, QUALCOMM
David Garcia, QUALCOMM
David Ligon, QUALCOMM
Jay Yun, QUALCOMM
Lee Howes, QUALCOMM
Richard Ruigrok, QUALCOMM
Robert J. Simpson, QUALCOMM
Sumesh Udayakumaran, QUALCOMM
Vineet Goel, QUALCOMM
Lihan Bin, QUALCOMM
Vlad Shimanskiy, QUALCOMM
Jian Liu, QUALCOMM
Tasneem Brutch, Samsung

Yoonseo Choi, Samsung
Dennis Adams, Sony
Pr-Anders Aronsson, Sony
Jim Rasmusson, Sony
Thierry Lepley, STMicroelectronics
Anton Gorenko, StreamHPC
Jakub Szuppe, StreamHPC
Vincent Hindriksen, StreamHPC
Alan Ward, Texas Instruments
Yuan Zhao, Texas Instruments
Pete Curry, Texas Instruments
Simon McIntosh-Smith, University of Bristol
James Price, University of Bristol
Paul Preney, University of Windsor
Shane Peelar, University of Windsor
Brian Hutsell, Vivante
Mike Cai, Vivante
Sumeet Kumar, Vivante
Wei-Lun Kao, Vivante
Xing Wang, Vivante
Jeff Fifield, Xilinx
Hem C. Neema, Xilinx
Henry Styles, Xilinx
Ralph Wittig, Xilinx
Ronan Keryell, Xilinx
AJ Guillon, YetiWare Inc

Chapter 6. The OpenCL C Programming Language

> *Note*
>
> This document starts at chapter 6 to keep the section numbers historically consistent with previous versions of the OpenCL and OpenCL C Programming Language specifications.

This section describes the OpenCL C programming language used to create kernels that are executed on OpenCL device(s). The OpenCL C programming language (also referred to as OpenCL C) is based on the ISO/IEC 9899:1999 C language Specification (a.k.a. “C99 Specification” or just “C99”) with specific extensions and restrictions. Please refer to that Specification for a detailed description of the language grammar. This document describes modifications and restrictions to C99 supported in OpenCL C.

In addition, some features of OpenCL C are based on the ISO/IEC 9899:2011 C language Specification (a.k.a. “C11 Specification” or just “C11”. Such features are described by reference to that Specification.

6.1. Supported Data Types

The following data types are supported.

6.1.1. Built-in Scalar Data Types

The following table describes the list of built-in scalar data types.

Table 1. Built-in Scalar Data Types

Type	Description
`bool`[1]	A conditional data type which is either *true* or *false*. The value *true* expands to the integer constant 1 and the value *false* expands to the integer constant 0.
`char`	A signed two’s complement 8-bit integer.
`unsigned char`, `uchar`	An unsigned 8-bit integer.
`short`	A signed two’s complement 16-bit integer.
`unsigned short`, `ushort`	An unsigned 16-bit integer.
`int`	A signed two’s complement 32-bit integer.
`unsigned int`, `uint`	An unsigned 32-bit integer.
`long`	A signed two’s complement 64-bit integer.
`unsigned long`, `ulong`	An unsigned 64-bit integer.

float	A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.
double[2]	A 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format.
half	A 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format.
size_t	The unsigned integer type[3] of the result of the sizeof operator.
ptrdiff_t	A signed integer type[3] that is the result of subtracting two pointers.
intptr_t	A signed integer type[3] with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer.
uintptr_t	An unsigned integer type[3] with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer.
void	The void type comprises an empty set of values; it is an incomplete type that cannot be completed.

[1] When any scalar value is converted to bool, the result is 0 if the value compares equal to 0; otherwise, the result is 1.

[2] The double scalar type is an optional type that is supported if the value of the CL_DEVICE_DOUBLE_FP_CONFIG device query is not zero.

[3] These are 32-bit types if the value of the CL_DEVICE_ADDRESS_BITS device query is 32-bits, and 64-bit types if the value of the query is 64-bits.

Most built-in scalar data types are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in scalar data type in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
bool	n/a
char	cl_char
unsigned char, uchar	cl_uchar
short	cl_short
unsigned short, ushort	cl_ushort

`int`	`cl_int`
`unsigned int`, `uint`	`cl_uint`
`long`	`cl_long`
`unsigned long`, `ulong`	`cl_ulong`
`float`	`cl_float`
`double`	`cl_double`
`half`	`cl_half`
`size_t`	n/a
`ptrdiff_t`	n/a
`intptr_t`	n/a
`uintptr_t`	n/a
`void`	`void`

The `half` data type

The `half` data type must be IEEE 754-2008 compliant. `half` numbers have 1 sign bit, 5 exponent bits, and 10 mantissa bits. The interpretation of the sign, exponent and mantissa is analogous to IEEE 754 floating-point numbers. The exponent bias is 15. The `half` data type must represent finite and normal numbers, denormalized numbers, infinities and NaN. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` using **vstore_half** and converting a `half` to a `float` using **vload_half** cannot be flushed to zero. Conversions from `float` to `half` correctly round the mantissa to 11 bits of precision. Conversions from `half` to `float` are lossless; all `half` numbers are exactly representable as `float` values.

The `half` data type can only be used to declare a pointer to a buffer that contains `half` values. A few valid examples are given below:

```
void
bar (__global half *p)
{
    ...
}

__kernel void
foo (__global half *pg, __local half *pl)
{
    __global half *ptr;
    int offset;

    ptr = pg + offset;
    bar(ptr);
}
```

Below are some examples that are not valid usage of the `half` type:

```
half a;
half b[100];
half *p;
a = *p; //  not allowed. must use *vload_half* function
```

Loads from a pointer to a `half` and stores to a pointer to a `half` can be performed using the vector data load and store functions **vload_half**, **vload_half*n***, **vloada_halfn** and **vstore_half**, **vstore_half*n***, and **vstorea_halfn**. The load functions read scalar or vector `half` values from memory and convert them to a scalar or vector `float` value. The store functions take a scalar or vector `float` value as input, convert it to a `half` scalar or vector value (with appropriate rounding mode) and write the `half` scalar or vector value to memory.

6.1.2. Built-in Vector Data Types[4]

The `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, and `float` vector data types are supported. The vector data type is defined with the type name, i.e. `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, or `float`, followed by a literal value *n* that defines the number of elements in the vector. Supported values of *n* are 2, 3, 4, 8, and 16 for all vector data types.

[4] Built-in vector data types are supported by the OpenCL implementation even if the underlying compute device does not support any or all of the vector data types. These are to be converted by the device compiler to appropriate instructions that use underlying built-in types supported natively by the compute device. Refer to Appendix B for a description of the order of the components of a vector type in memory.

The following table describes the list of built-in vector data types.

Table 2. Built-in Vector Data Types

Type	**Description**
`charn`	A vector of *n* 8-bit signed two's complement integer values.
`ucharn`	A vector of *n* 8-bit unsigned integer values.
`shortn`	A vector of *n* 16-bit signed two's complement integer values.
`ushortn`	A vector of *n* 16-bit unsigned integer values.
`intn`	A vector of *n* 32-bit signed two's complement integer values.
`uintn`	A vector of *n* 32-bit unsigned integer values.
`longn`	A vector of *n* 64-bit signed two's complement integer values.
`ulongn`	A vector of *n* 64-bit unsigned integer values.
`floatn`	A vector of *n* 32-bit floating-point values.
`doublen`[5]	A vector of *n* 64-bit floating-point values.

[5] The `double` vector type is an optional type that is supported if the value of the

CL_DEVICE_DOUBLE_FP_CONFIG device query is not zero.

The built-in vector data types are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in vector data type in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
charn	cl_charn
ucharn	cl_ucharn
shortn	cl_shortn
ushortn	cl_ushortn
intn	cl_intn
uintn	cl_uintn
longn	cl_longn
ulongn	cl_ulongn
floatn	cl_floatn
doublen	cl_doublen

6.1.3. Other Built-in Data Types

The following table describes the list of additional data types supported by OpenCL.

Table 3. Other Built-in Data Types

Type	Description
image2d_t	A 2D image[6].
image3d_t	A 3D image[6].
image2d_array_t	A 2D image array[6].
image1d_t	A 1D image[6].
image1d_buffer_t	A 1D image created from a buffer object[6].
image1d_array_t	A 1D image array[6].
image2d_depth_t	A 2D depth image[6].
image2d_array_depth_t	A 2D depth image array[6].
sampler_t	A sampler type[6].
queue_t	A device command queue. This queue can only be used to enqueue commands from kernels executing on the device.
ndrange_t	The N-dimensional range over which a kernel executes.
clk_event_t	A device side event that identifies a command enqueue to a device command queue.

`reserve_id_t`	A reservation ID. This opaque type is used to identify the reservation for reading and writing a pipe.
`event_t`	An event. This can be used to identify async copies from `global` to `local` memory and vice-versa.

`cl_mem_fence_flags`	This is a bitfield and can be 0 or a combination of the following values ORed together:
	
	`CLK_GLOBAL_MEM_FENCE`
	`CLK_LOCAL_MEM_FENCE`
	`CLK_IMAGE_MEM_FENCE`
	
	These flags are described in detail in the synchronization functions section.

[6] Refer to the detailed description of the built-in functions that use this type.

> The `image2d_t`, `image3d_t`, `image2d_array_t`, `image1d_t`, `image1d_buffer_t`, `image1d_array_t`, `image2d_depth_t`, `image2d_array_depth_t` and `sampler_t` types are only defined if the device supports images, i.e. the value of the `CL_DEVICE_IMAGE_SUPPORT` device query) is `CL_TRUE`.

The C99 derived types (arrays, structs, unions, functions, and pointers), constructed from the built-in scalar, vector, and other data types are supported, with specified restrictions.

The following tables describe the other built-in data types in OpenCL described in Other Built-in Data Types and the corresponding data type available to the application:

Type in OpenCL C	API type for application
`queue_t`	`cl_command_queue`
`clk_event_t`	`cl_event`

6.1.4. Reserved Data Types

The data type names described in the following table are reserved and cannot be used by applications as type names. The vector data type names defined in Built-in Vector Data Types, but where *n* is any value other than 2, 3, 4, 8 and 16, are also reserved.

Table 4. Reserved Data Types

Type	Description
`bool`*n*	A boolean vector.
`half`*n*	A 16-bit floating-point vector.
`quad`, `quad`*n*	A 128-bit floating-point scalar and vector.
`complex half`, `complex half`*n*	A complex 16-bit floating-point scalar and vector.

imaginary half, imaginary half*n*	An imaginary 16-bit floating-point scalar and vector.
complex float, complex float*n*	A complex 32-bit floating-point scalar and vector.
imaginary float, imaginary float*n*	An imaginary 32-bit floating-point scalar and vector.
complex double, complex double*n*	A complex 64-bit floating-point scalar and vector.
imaginary double, imaginary double*n*	An imaginary 64-bit floating-point scalar and vector.
complex quad, complex quad*n*	A complex 128-bit floating-point scalar and vector.
imaginary quad, imaginary quad*n*	An imaginary 128-bit floating-point scalar and vector.
float*n*x*m*	An $n \times m$ matrix of single precision floating-point values stored in column-major order.
double*n*x*m*	An $n \times m$ matrix of double precision floating-point values stored in column-major order.
long double long double*n*	A floating-point scalar and vector type with at least as much precision and range as a double and no more precision and range than a quad.
long long, long long*n*	A 128-bit signed integer scalar and vector.
unsigned long long, ulong long, ulong long*n*	A 128-bit unsigned integer scalar and vector.

6.1.5. Alignment of Types

A data item declared to be a data type in memory is always aligned to the size of the data type in bytes. For example, a `float4` variable will be aligned to a 16-byte boundary, a `char2` variable will be aligned to a 2-byte boundary.

For 3-component vector data types, the size of the data type is `4 * sizeof(component)`. This means that a 3-component vector data type will be aligned to a `4 * sizeof(component)` boundary. The **vload3** and **vstore3** built-in functions can be used to read and write, respectively, 3-component vector data types from an array of packed scalar data type.

A built-in data type that is not a power of two bytes in size must be aligned to the next larger power of two. This rule applies to built-in types only, not structs or unions.

The OpenCL compiler is responsible for aligning data items to the appropriate alignment as required by the data type. For arguments to a `__kernel` function declared to be a pointer to a data type, the OpenCL compiler can assume that the pointee is always appropriately aligned as required by the data type. The behavior of an unaligned load or store is undefined, except for the vector data load and store functions **vload*n***, **vload_half*n***, **vstore*n***, and **vstore_half*n***. The vector load functions can read a vector from an address aligned to the element type of the vector. The vector store functions can write a vector to an address aligned to the element type of the vector.

6.1.6. Vector Literals

Vector literals can be used to create vectors from a list of scalars, vectors or a mixture thereof. A vector literal can be used either as a vector initializer or as a primary expression. A vector literal cannot be used as an l-value.

A vector literal is written as a parenthesized vector type followed by a parenthesized comma delimited list of parameters. A vector literal operates as an overloaded function. The forms of the function that are available is the set of possible argument lists for which all arguments have the same element type as the result vector, and the total number of elements is equal to the number of elements in the result vector. In addition, a form with a single scalar of the same type as the element type of the vector is available. For example, the following forms are available for `float4`:

```
(float4)( float, float, float, float )
(float4)( float2, float, float )
(float4)( float, float2, float )
(float4)( float, float, float2 )
(float4)( float2, float2 )
(float4)( float3, float )
(float4)( float, float3 )
(float4)( float )
```

Operands are evaluated by standard rules for function evaluation, except that implicit scalar widening shall not occur. The order in which the operands are evaluated is undefined. The operands are assigned to their respective positions in the result vector as they appear in memory order. That is, the first element of the first operand is assigned to `result.x`, the second element of the first operand (or the first element of the second operand if the first operand was a scalar) is assigned to `result.y`, etc. In the case of the form that has a single scalar operand, the operand is replicated across all lanes of the vector.

Examples:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uint4 u = (uint4)(1); //  u will be (1, 1, 1, 1).
float4 f = (float4)<a name="__indexterm-37847560" type="indexterm">  </a>float2)(1.0f,
2.0f), (float2)(3.0f, 4.0f;
float4 f = (float4)(1.0f, (float2)(2.0f, 3.0f), 4.0f);
float4 f = (float4)(1.0f, 2.0f); //  error
```

6.1.7. Vector Components

The components of vector data types with 1 ... 4 components can be addressed as `<vector_data_type>.xyzw`. Vector data types of type `char2`, `uchar2`, `short2`, `ushort2`, `int2`, `uint2`, `long2`, `ulong2`, and `float2` can access `.xy` elements. Vector data types of type `char3`, `uchar3`, `short3`, `ushort3`, `int3`, `uint3`, `long3`, `ulong3`, and `float3` can access `.xyz` elements. Vector data types of type `char4`, `uchar4`, `short4`, `ushort4`, `int4`, `uint4`, `long4`, `ulong4`, and `float4` can access `.xyzw` elements.

Accessing components beyond those declared for the vector type is an error so, for example:

```
float2 pos;
pos.x = 1.0f; // is legal
pos.z = 1.0f; // is illegal

float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names after the period (.).

```
float4 c;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyz = (float3)(3.0f, 4.0f, 5.0f);
```

The component selection syntax also allows components to be permuted or replicated.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 swiz= pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)

float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

The component group notation can occur on the left hand side of an expression. To form an l-value, swizzling must be applied to an l-value of vector type, contain no duplicate components, and it results in an l-value of scalar or vector type, depending on number of components specified. Each component must be a supported scalar or vector type.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

pos.xw = (float2)(5.0f, 6.0f);// pos = (5.0f, 2.0f, 3.0f, 6.0f)
pos.wx = (float2)(7.0f, 8.0f);// pos = (8.0f, 2.0f, 3.0f, 7.0f)
pos.xyz = (float3)(3.0f, 5.0f, 9.0f); // pos = (3.0f, 5.0f, 9.0f, 4.0f)
pos.xx = (float2)(3.0f, 4.0f);// illegal - 'x' used twice

// illegal - mismatch between float2 and float4
pos.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 a, b, c, d;
float16 x;
x = (float16)(a, b, c, d);
x = (float16)(a.xxxx, b.xyz, c.xyz, d.xyz, a.yzw);

// illegal - component a.xxxxxxx is not a valid vector type
x = (float16)(a.xxxxxxx, b.xyz, c.xyz, d.xyz);
```

Elements of vector data types can also be accessed using a numeric index to refer to the appropriate element in the vector. The numeric indices that can be used are given in the table below:

Table 5. Numeric indices for built-in vector data types

Vector Components	Numeric indices that can be used
2-component	0, 1
3-component	0, 1, 2
4-component	0, 1, 2, 3
8-component	0, 1, 2, 3, 4, 5, 6, 7
16-component	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

The numeric indices must be preceded by the letter `s` or `S`.

In the following example

```
float8 f;
```

`f.s0` refers to the 1st element of the `float8` variable `f` and `f.s7` refers to the 8th element of the `float8` variable `f`.

In the following example

```
float16 x;
```

`x.sa` (or `x.sA`) refers to the 11th element of the `float16` variable `x` and `x.sf` (or `x.sF`) refers to the 16th element of the `float16` variable `x`.

The numeric indices used to refer to an appropriate element in the vector cannot be intermixed with `.xyzw` notation used to access elements of a 1 .. 4 component vector.

For example

```
float4 f, a;

a = f.x12w;       // illegal use of numeric indices with .xyzw

a.xyzw = f.s0123; // valid
```

Vector data types can use the `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes to get smaller vector types or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes can be used until they refer to a scalar term.

The `.lo` suffix refers to the lower half of a given vector. The `.hi` suffix refers to the upper half of a given vector.

The `.even` suffix refers to the even elements of a vector. The `.odd` suffix refers to the odd elements of a vector.

Some examples to help illustrate this are given below:

```
float4 vf;

float2 low = vf.lo;    // returns vf.xy
float2 high = vf.hi;   // returns vf.zw

float2 even = vf.even; // returns vf.xz
float2 odd = vf.odd;   // returns vf.yw
```

The suffixes `.lo` (or `.even`) and `.hi` (or `.odd`) for a 3-component vector type operate as if the 3-component vector type is a 4-component vector type with the value in the `w` component undefined.

Some examples are given below:

```
float8 vf;
float4 odd = vf.odd;
float4 even = vf.even;
float2 high = vf.even.hi;
float2 low = vf.odd.lo;

// interleave LR stereo stream
float4 left, right;
float8 interleaved;
interleaved.even = left;
interleaved.odd = right;

// deinterleave
left = interleaved.even;
right = interleaved.odd;

// transpose a 4x4 matrix

void transpose( float4 m[4] )
{
    // read matrix into a float16 vector
    float16 x = (float16)( m[0], m[1], m[2], m[3] );
    float16 t;

    // transpose
    t.even = x.lo;
    t.odd = x.hi;
    x.even = t.lo;
    x.odd = t.hi;

    // write back
    m[0] = x.lo.lo; // { m[0][0], m[1][0], m[2][0], m[3][0] }
    m[1] = x.lo.hi; // { m[0][1], m[1][1], m[2][1], m[3][1] }
    m[2] = x.hi.lo; // { m[0][2], m[1][2], m[2][2], m[3][2] }
    m[3] = x.hi.hi; // { m[0][3], m[1][3], m[2][3], m[3][3] }
}

float3 vf = (float3)(1.0f, 2.0f, 3.0f);
float2 low = vf.lo; // (1.0f, 2.0f);
float2 high = vf.hi; // (3.0f, _undefined_);
```

It is an error to take the address of a vector element and will result in a compilation error. For example:

```
float8 vf;

float *f = &vf.x; m         // is illegal
float2 *f2 = &vf.s07;       // is illegal

float4 *odd = &vf.odd;      // is illegal
float4 *even = &vf.even;    // is illegal
float2 *high = &vf.even.hi; // is illegal
float2 *low = &vf.odd.lo;   // is illegal
```

6.1.8. Aliasing Rules

OpenCL C programs shall comply with the C99 type-based aliasing rules defined in section 6.5, item 7 of the C99 Specification. The OpenCL C built-in vector data types are considered aggregate[7] types for the purpose of applying these aliasing rules.

[7] That is, for the purpose of applying type-based aliasing rules, a built-in vector data type will be considered equivalent to the corresponding array type.

6.1.9. Keywords

The following names are reserved for use as keywords in OpenCL C and shall not be used otherwise.

- Names reserved as keywords by C99.
- OpenCL C data types defined in Built-in Vector Data Types, Other Built-in Data Types, and Reserved Data Types.
- Address space qualifiers: `__global`, `global`, `__local`, `local`, `__constant`, `constant`, `__private`, and `private`. `__generic` and `generic` are reserved for future use.
- Function qualifiers: `__kernel` and `kernel`.
- Access qualifiers: `__read_only`, `read_only`, `__write_only`, `write_only`, `__read_write` and `read_write`.
- `uniform`, `pipe`.

6.2. Conversions and Type Casting

6.2.1. Implicit Conversions

Implicit conversions between scalar built-in types defined in Built-in Scalar Data Types (except `void` and `half`[8]) are supported. When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value 5 will be converted to the floating-point value 5.0.

[8] Unless the **cl_khr_fp16** extension is supported and has been enabled.

Implicit conversions from a scalar type to a vector type are allowed. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector. The scalar type is then widened to the vector.

Implicit conversions between built-in vector data types are disallowed.

Implicit conversions for pointer types follow the rules described in the <C99-spec,C99 Specification>>.

6.2.2. Explicit Casts

Standard typecasts for built-in scalar data types defined in Built-in Scalar Data Types will perform appropriate conversion (except `void` and `half`[9]). In the example below:

[9] Unless the **cl_khr_fp16** extension is supported and has been enabled.

```
float f = 1.0f;
int i = (int)f;
```

`f` stores `0x3F800000` and `i` stores `0x1` which is the floating-point value `1.0f` in `f` converted to an integer value.

Explicit casts between vector types are not legal. The examples below will generate a compilation error.

```
int4 i;
uint4 u = (uint4) i; //  not allowed

float4 f;
int4 i = (int4) f; //  not allowed

float4 f;
int8 i = (int8) f; //  not allowed
```

Scalar to vector conversions may be performed by casting the scalar to the desired vector data type. Type casting will also perform appropriate arithmetic conversion. The round to zero rounding mode will be used for conversions to built-in integer vector types. The default rounding mode will be used for conversions to floating-point vector types. When casting a `bool` to a vector integer data type, the vector components will be set to -1 (i.e. all bits set) if the bool value is *true* and 0 otherwise.

Below are some correct examples of explicit casts.

```
float f = 1.0f;
float4 va = (float4)f;

// va is a float4 vector with elements (f, f, f, f).

uchar u = 0xFF;
float4 vb = (float4)u;

// vb is a float4 vector with elements
// ((float)u, (float)u, (float)u, (float)u).

float f = 2.0f;
int2 vc = (int2)f;

// vc is an int2 vector with elements ((int)f, (int)f).

uchar4 vtrue = (uchar4)true;

// vtrue is a uchar4 vector with elements (0xff, 0xff,
// 0xff, 0xff).
```

6.2.3. Explicit Conversions

Explicit conversions may be performed using the

```
convert_destType(sourceType)
```

suite of functions. These provide a full set of type conversions between supported scalar, vector, and other data types except for the following types: `bool`, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, and `void`.

The number of elements in the source and destination vectors must match.

In the example below:

```
uchar4 u;
int4 c = convert_int4(u);
```

`convert_int4` converts a `uchar4` vector `u` to an `int4` vector `c`.

```
float f;
int i = convert_int(f);
```

`convert_int` converts a `float` scalar `f` to an int scalar `i`.

The behavior of the conversion may be modified by one or two optional modifiers that specify saturation for out-of-range inputs and rounding behavior.

The full form of the scalar convert function is:

```
destType *convert_destType<_sat><_roundingMode>*(sourceType)
```

The full form of the vector convert function is:

```
destType__n__ *convert_destType__n__<_sat><_roundingMode>*(sourceType__n__)
```

Data Types

Conversions are available for the following scalar types: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, and built-in vector types derived therefrom. The operand and result type must have the same number of elements. The operand and result type may be the same type in which case the conversion has no effect on the type or value of an expression.

Conversions between integer types follow the conversion rules specified in sections 6.3.1.1 and 6.3.1.3 of the C99 Specification except for out-of-range behavior and saturated conversions.

Rounding Modes

Conversions to and from floating-point type shall conform to IEEE-754 rounding rules. Conversions may have an optional rounding mode modifier described in the following table.

Table 6. Rounding Modes

Modifier	Rounding Mode Description
`_rte`	Round to nearest even
`_rtz`	Round toward zero
`_rtp`	Round toward positive infinity
`_rtn`	Round toward negative infinity
no modifier specified	Use the default rounding mode for this destination type, `_rtz` for conversion to integers or the default rounding mode for conversion to floating-point types.

By default, conversions to integer type use the `_rtz` (round toward zero) rounding mode and conversions to floating-point type[10] use the default rounding mode. The only default floating-point rounding mode supported is round to nearest even i.e the default rounding mode will be `_rte` for floating-point types.

[10] For conversions to floating-point format, when a finite source value exceeds the maximum representable finite floating-point destination value, the rounding mode will affect whether the result is the maximum finite floating-point value or infinity of same sign as the source value, per IEEE-754 rules for rounding.

Out-of-Range Behavior and Saturated Conversions

When the conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out-of-range. The result of out-of-range conversion is determined by the conversion rules specified by section 6.3 of the C99 Specification. When converting from a floating-point type to integer type, the behavior is implementation-defined.

Conversions to integer type may opt to convert using the optional saturated mode by appending the _sat modifier to the conversion function name. When in saturated mode, values that are outside the representable range shall clamp to the nearest representable value in the destination format. (NaN should be converted to 0).

Conversions to floating-point type shall conform to IEEE-754 rounding rules. The `_sat` modifier may not be used for conversions to floating-point formats.

Explicit Conversion Examples

Example 1:

```
short4 s;

// negative values clamped to 0
ushort4 u = convert_ushort4_sat( s );

// values > CHAR_MAX converted to CHAR_MAX
// values < CHAR_MIN converted to CHAR_MIN
char4 c = convert_char4_sat( s );
```

Example 2:

```
float4 f;

// values implementation defined for
// f > INT_MAX, f < INT_MIN or NaN
int4 i = convert_int4( f );

// values > INT_MAX clamp to INT_MAX, values < INT_MIN clamp
// to INT_MIN. NaN should produce 0.
// The _rtz_ rounding mode is used to produce the integer values.
int4 i2 = convert_int4_sat( f );

// similar to convert_int4, except that floating-point values
// are rounded to the nearest integer instead of truncated
int4 i3 = convert_int4_rte( f );

// similar to convert_int4_sat, except that floating-point values
// are rounded to the nearest integer instead of truncated
int4 i4 = convert_int4_sat_rte( f );
```

Example 3:

```
int4 i;

// convert ints to floats using the default rounding mode.
float4 f = convert_float4( i );

// convert ints to floats. integer values that cannot
// be exactly represented as floats should round up to the
// next representable float.
float4 f = convert_float4_rtp( i );
```

6.2.4. Reinterpreting Data As Another Type

It is frequently necessary to reinterpret bits in a data type as another data type in OpenCL. This is typically required when direct access to the bits in a floating-point type is needed, for example to mask off the sign bit or make use of the result of a vector relational operator on floating-point data[11]. Several methods to achieve this (non-) conversion are frequently practiced in C, including pointer aliasing, unions and memcpy. Of these, only memcpy is strictly correct in C99. Since OpenCL does not provide **memcpy**, other methods are needed.

[11] In addition, some other extensions to the C language designed to support particular vector ISA (e.g. AltiVec™, CELL Broadband Engine™ Architecture) use such conversions in conjunction with swizzle operators to achieve type unconversion. So as to support legacy code of this type, **as_typen**() allows conversions between vectors of the same size but different numbers of elements, even though the behavior of this sort of conversion is not likely to be portable except to other OpenCL implementations for the same hardware architecture. AltiVec is a trademark of Motorola Inc. Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

Reinterpreting Types Using Unions

The OpenCL language extends the union to allow the program to access a member of a union object using a member of a different type. The relevant bytes of the representation of the object are treated as an object of the type used for the access. If the type used for access is larger than the representation of the object, then the value of the additional bytes is undefined.

Examples:

```
// d only if double precision is supported
union { float f; uint u; double d; } u;

u.u = 1;    // u.f contains 2**-149.  u.d is undefined --
            // depending on endianness the low or high half
            // of d is unknown

u.f = 1.0f; // u.u contains 0x3f800000, u.d contains an
            // undefined value -- depending on endianness
            // the low or high half of d is unknown

u.d = 1.0;  // u.u contains 0x3ff00000 (big endian) or 0
            // (little endian). u.f contains either 0x1.ep0f
            // (big endian) or 0.0f (little endian)
```

Reinterpreting Types Using as_type() and as_type*n*()

All data types described in Built-in Scalar Data Types and Built-in Vector Data Types (except `bool`, `half`[12] and `void`) may be also reinterpreted as another data type of the same size using the **as_type**() operator for scalar data types and the **as_type*n***() operator[13] for vector data types. When the operand and result type contain the same number of elements, the bits in the operand shall be returned directly without modification as the new type. The usual type promotion for function arguments shall not be performed.

[12] Unless the **cl_khr_fp16** extension is supported and has been enabled.

[13] While the union is intended to reflect the organization of data in memory, the **as_type**() and **as_type*n***() constructs are intended to reflect the organization of data in register. The **as_type**() and **as_type*n***() constructs are intended to compile to no instructions on devices that use a shared register file designed to operate on both the operand and result types. Note that while differences in memory organization are expected to largely be limited to those arising from endianness, the register based representation may also differ due to size of the element in register. (For example, an architecture may load a `char` into a 32-bit register, or a `char` vector into a SIMD vector register with fixed 32-bit element size.) If the element count does not match, then the implementation should pick a data representation that most closely matches what would happen if an appropriate result type operator was applied to a register containing data of the source type. If the number of elements matches, then the **as_type*n***() should faithfully reproduce the behavior expected from a similar data type reinterpretation using memory/unions. So, for example if an implementation stores all single precision data as `double` in register, it should implement **as_int**(`float`) by first downconverting the `double` to single precision and then (if necessary) moving the single precision

bits to a register suitable for operating on integer data. If data stored in different address spaces do not have the same endianness, then the “dominant endianness” of the device should prevail.

For example, `as_float(0x3f800000)` returns `1.0f`, which is the value that the bit pattern `0x3f800000` has if viewed as an IEEE-754 single precision value.

When the operand and result type contain a different number of elements, the result shall be implementation-defined except if the operand is a 4-component vector and the result is a 3-component vector. In this case, the bits in the operand shall be returned directly without modification as the new type. That is, a conforming implementation shall explicitly define a behavior, but two conforming implementations need not have the same behavior when the number of elements in the result and operand types does not match. The implementation may define the result to contain all, some or none of the original bits in whatever order it chooses. It is an error to use **as_type**() or **as_type*n***() operator to reinterpret data to a type of a different number of bytes.

Examples:

```
float f = 1.0f;
uint u = as_uint(f); // Legal. Contains:  0x3f800000

float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
// Legal. Contains:
// (int4)(0x3f800000, 0x40000000, 0x40400000, 0x40800000)
int4 i = as_int4(f);

float4 f, g;
int4  is_less = f < g;

// Legal. f[i] = f[i] < g[i] ? f[i] : 0.0f
f = as_float4(as_int4(f) & is_less);

int i;
// Legal. Result is implementation-defined.
short2 j = as_short2(i);

int4 i;
// Legal. Result is implementation-defined.
short8 j = as_short8(i);

float4 f;
// Error. Result and operand have different sizes
double4 g = as_double4(f); // Only if double precision is supported.

float4 f;
// Legal. g.xyz will have same values as f.xyz. g.w is undefined
float3 g = as_float3(f);
```

6.2.5. Pointer Casting

Pointers to old and new types may be cast back and forth to each other. Casting a pointer to a new type represents an unchecked assertion that the address is correctly aligned. The developer will also need to know the endianness of the OpenCL device and the endianness of the data to determine how the scalar and vector data elements are stored in memory.

6.2.6. Usual Arithmetic Conversions

Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. For this purpose, all vector types shall be considered to have higher conversion ranks than scalars. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the usual arithmetic conversions. If the operands are of more than one vector type, then an error shall occur. Implicit conversions between vector types are not permitted.

Otherwise, if there is only a single vector type, and all other operands are scalar types, the scalar types are converted to the type of the vector element, then widened into a new vector containing the same number of elements as the vector, by duplication of the scalar value across the width of the new vector. An error shall occur if any scalar operand has greater rank than the type of the vector element. For this purpose, the rank order defined as follows:

1. The rank of a floating-point type is greater than the rank of another floating-point type, if the first floating-point type can exactly represent all numeric values in the second floating-point type. (For this purpose, the encoding of the floating-point value is used, rather than the subset of the encoding usable by the device.)
2. The rank of any floating-point type is greater than the rank of any integer type.
3. The rank of an integer type is greater than the rank of an integer type with less precision.
4. The rank of an unsigned integer type is **greater than** the rank of a signed integer type with the same precision[14].
5. The rank of the bool type is less than the rank of any other type.
6. The rank of an enumerated type shall equal the rank of the compatible integer type.
7. For all types, `T1`, `T2` and `T3`, if `T1` has greater rank than `T2`, and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

[14] This is different from the standard integer conversion rank described in section 6.3.1.1 of the C99 Specification

Otherwise, if all operands are scalar, the usual arithmetic conversions apply, per section 6.3.1.8 of the C99 Specification.

Both the standard orderings in sections 6.3.1.8 and 6.3.1.1 of the C99 Specification were examined and rejected. Had we used integer conversion rank here, `int4 + 0U` would have been legal and had `int4` return type. Had we used standard C99 usual arithmetic conversion rules for scalars, then the standard integer promotion would have been performed on vector integer element types and `short8 + char` would either have return type of `int8` or be illegal.

6.3. Operators

6.3.1. Arithmetic Operators

The arithmetic operators add (+), subtract (-), multiply (*) and divide (/) operate on built-in integer and floating-point scalar, and vector data types. The remainder (%) operates on built-in integer scalar and integer vector data types. All arithmetic operators return result of the same built-in type (integer or floating-point) as the type of the operands, after operand type conversion. After conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal. Division on integer types which results in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type will not cause an exception but will result in an unspecified value. A divide by zero with integer types does not cause an exception but will result in an unspecified value. Division by zero for floating-point types will result in ±∞ or NaN as prescribed by the IEEE-754 standard. Use the built-in functions **dot** and **cross** to get, respectively, the vector dot product and the vector cross product.

6.3.2. Unary Operators

The arithmetic unary operators (+ and -) operate on built-in scalar and vector types.

6.3.3. Operators

The arithmetic post- and pre-increment and decrement operators (-- and ++) operate on built-in scalar and vector types except the built-in scalar and vector `float` types[15]. All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.

[15] The pre- and post- increment operators may have unexpected behavior on floating-point values and are therefore not supported for floating-point scalar and vector built-in types. For example, if variable *a* has type `float` and holds the value `0x1.0p25f`, then `a++` returns `0x1.0p25f`. Also, `(a++)--` is not guaranteed to return *a*, if *a* has fractional value. In non-default rounding modes, `(a++)--` may produce the same result as `a++` or `a--` for large *a*.

6.3.4. Relational Operators

The relational operators[16] greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate on scalar and vector types. All relational operators result in an integer type. After operand type conversion, the following cases are valid:

[16] To test whether any or all elements in the result of a vector relational operator test *true*, for example to use in the context in an **if ( )** statement, please see the **any** and **all** builtins.

- The two operands are scalars. In this case, the operation is applied, resulting in an `int` scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result. For scalar types, the relational operators shall return 0 if the specified relation is *false* and 1 if the specified relation is *true*. For vector types, the relational operators shall return 0 if the specified relation is *false* and -1 (i.e. all bits set) if the specified relation is *true*. The relational operators always return 0 if either argument is not a number (NaN).

6.3.5. Equality Operators

The equality operators[17] equal (==) and not equal (**!**=) operate on built-in scalar and vector types. All equality operators result in an integer type. After operand type conversion, the following cases are valid:

[17] To test whether any or all elements in the result of a vector equality operator test true, for example to use in the context in an **if ( )** statement, please see the **any** and **all** builtins.

- The two operands are scalars. In this case, the operation is applied, resulting in a scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar may be subject to the

usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the equality operators return 0 if the specified relation is *false* and return 1 if the specified relation is *true*. For vector types, the equality operators shall return 0 if the specified relation is *false* and -1 (i.e. all bits set) if the specified relation is *true*. The equality operator equal (==) returns 0 if one or both arguments are not a number (NaN). The equality operator not equal (**!**=) returns 1 (for scalar source operands) or -1 (for vector source operands) if one or both arguments are not a number (NaN).

6.3.6. Bitwise Operators

The bitwise operators and (**&**), or (|), exclusive or (^), and not (~) operate on all scalar and vector built-in types except the built-in scalar and vector `float` types. For vector built-in types, the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

6.3.7. Logical Operators

The logical operators and (**&&**) and or (||) operate on all scalar and vector built-in types. For scalar built-in types only, and (**&&**) will only evaluate the right hand operand if the left hand operand compares unequal to 0. For scalar built-in types only, or (||) will only evaluate the right hand operand if the left hand operand compares equal to 0. For built-in vector types, both operands are evaluated and the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

The logical operator exclusive or (^^) is reserved.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the logical operators shall return 0 if the result of the operation is *false* and 1 if the result is *true*. For vector types, the logical operators shall return 0 if the result of the operation is *false* and -1 (i.e. all bits set) if the result is *true*.

6.3.8. Unary Logical Operator

The logical unary operator not (**!**) operates on all scalar and vector built-in types. For built-in vector types, the operators are applied component-wise.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the result of the logical unary operator is 0 if the value of its operand compares unequal to 0, and 1 if the value of its operand compares equal to 0. For vector types, the unary operator shall return a 0 if the value of its operand compares unequal to 0, and -1 (i.e. all bits set) if the value of its operand compares equal to 0.

6.3.9. Ternary Selection Operator

The ternary selection operator (**?:**) operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression *exp1*, which can be a scalar or vector result except `float`. If all three expressions are scalar values, the C99 rules for ternary operator are followed. If the result is a vector value, then this is equivalent to calling **select**(*exp3*, *exp2*, *exp1*). The **select** function is described in Scalar and Vector Relational Functions. The second and third expressions can be any type, as long their types match, or there is an implicit conversion that can be applied to one of the expressions to make their types match, or one is a vector and the other is a scalar and the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand and widened to the same type as the vector type. This resulting matching type is the type of the entire expression.

6.3.10. Shift Operators

The operators right-shift (>>), left-shift (<<) operate on all scalar and vector built-in types except the built-in scalar and vector `float` types. For built-in vector types, the operators are applied component-wise. For the right-shift (>>), left-shift (<<) operators, the rightmost operand must be a scalar if the first operand is a scalar, and the rightmost operand can be a vector or scalar if the first operand is a vector.

The result of `E1` << `E2` is `E1` left-shifted by $\log_2(N)$ least significant bits in `E2` viewed as an unsigned integer value, where N is the number of bits used to represent the data type of `E1` after integer promotion[18], if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. The vacated bits are filled with zeros.

[18] Integer promotion is described in section 6.3.1.1 of the C99 Specification.

The result of `E1` >> `E2` is `E1` right-shifted by $\log_2(N)$ least significant bits in `E2` viewed as an unsigned integer value, where N is the number of bits used to represent the data type of `E1` after integer promotion, if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the vacated bits are filled with zeros. If `E1` has a signed type and a negative value, the vacated bits are filled with ones.

6.3.11. Sizeof Operator

The `sizeof` operator yields the size (in bytes) of its operand, including any padding bytes needed for alignment, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is of type `size_t`. If the type of the operand is a variable length array[19] type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

[19] Variable length arrays are not supported in OpenCL C.

When applied to an operand that has type `char` or `uchar`, the result is 1. When applied to an operand that has type `short`, `ushort`, or `half` the result is 2. When applied to an operand that has type `int`, `uint` or `float`, the result is 4. When applied to an operand that has type `long`, `ulong` or `double`, the result is 8. When applied to an operand that is a vector type, the result[20] is number of components * size of each scalar component. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. The `sizeof` operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field struct member[21].

[20] Except for 3-component vectors whose size is defined as 4 * size of each scalar component.

[21] Bit-field struct members are not supported in OpenCL C.

The behavior of applying the `sizeof` operator to the `bool`, `image2d_t`, `image3d_t`, `image2d_array_t`, `image2d_depth_t`, `image2d_array_depth_t`, `image1d_t`, `image1d_buffer_t` or `image1d_array_t`, `sampler_t`, `clk_event_t`, `queue_t` and `event_t` types is implementation-defined.

6.3.12. Comma Operator

The comma (,) operator operates on expressions by returning the type and value of the right-most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right.

6.3.13. Indirection Operator

The unary (*****) operator denotes indirection. If the operand points to an object, the result is an l-value designating the object. If the operand has type "pointer to *type*", the result has type "*type*". If an invalid value has been assigned to the pointer, the behavior of the unary ***** operator is undefined[22].

[22] Among the invalid values for dereferencing a pointer by the unary ***** operator are a null

pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime. If **P** is an l-value and **T** is the name of an object pointer type, **(T)P** is an l-value that has a type compatible with that to which **T** points.

6.3.14. Address Operator

The unary (**&**) operator returns the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*". If the operand is the result of a unary ***** operator, neither that operator nor the **&** operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an l-value. Similarly, if the operand is the result of a **[]** operator, neither the **&** operator nor the unary ***** that is implied by the **[]** is evaluated and the result is as if the **&** operator were removed and the **[]** operator were changed to a + operator. Otherwise, the result is a pointer to the object designated by its operand[23].

[23] Thus, **&*E** is equivalent to **E** (even if **E** is a null pointer), and **&(E1[E2])** to **((E1)+ (E2))**. It is always true that if **E** is an l-value that is a valid operand of the unary **&** operator, **&E** is an l-value equal to **E**.

6.3.15. Assignment Operator

Assignments of values to variable names are done with the assignment operator (=), like

> *lvalue* = *expression*

The assignment operator stores the value of *expression* into *lvalue*. The *expression* and *lvalue* must have the same type, or the expression must have a type in Built-in Scalar Data Types, in which case an implicit conversion will be done on the expression before the assignment is done.

If *expression* is a scalar type and *lvalue* is a vector type, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

Any other desired type-conversions must be specified explicitly. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[]**) are all l-values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value.

The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined. Other assignment operators are the assignments add into (+=), subtract from (**-**=), multiply into (=), divide into (/=), modulus into (**%**=), left shift by (<<=), right shift by (>>=), and into (**&**=), inclusive or into (|=), and exclusive or into (^=).

The expression

> *lvalue op*=* expression*

is equivalent to

lvalue = *lvalue op expression*

and the *lvalue* and *expression* must satisfy the requirements for both operator *op* and assignment (=).

Except for the `sizeof` operator, the `half` data type cannot be used with any of the operators described in this section.

6.4. Vector Operations

Vector operations are component-wise. Usually, when an operator operates on a vector, it is operating independently on each component of the vector, in a component-wise fashion.

For example,

```
float4 v, u;
float f;

v = u + f;
```

will be equivalent to

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
v.w = u.w + f;
```

And

```
float4 v, u, w;

w = v + u;
```

will be equivalent to

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

and likewise for most operators and all integer and floating-point vector types.

6.5. Address Space Qualifiers

OpenCL implements the following disjoint named address spaces: `__global`, `__local`, `__constant` and `__private`. The address space qualifier may be used in variable declarations to specify the region of memory that is used to allocate the object. The C syntax for type qualifiers is extended in OpenCL to include an address space name as a valid type qualifier. If the type of an object is qualified by an address space name, the object is allocated in the specified address name; otherwise, the object is allocated in the generic address space.

The address space names without the `__` prefix, i.e. `global`, `local`, `constant` and `private`, may be substituted for the corresponding address space names with the `__` prefix.

The address space name for arguments to a function in a program, or local variables of a function is `__private`. All function arguments shall be in the `__private` address space. The address space for a variable at program scope, a `static` or `extern` variable inside a function can either be `__global` or `__constant`, but defaults to `__global` if not specified.

Examples:

```
// declares a pointer p in the private address space that
// points to an object in address space global
global int *p;

void foo (...)
{
    // declares an array of 4 floats in the private address space
    float x[4];
    ...
}
```

OpenCL 2.0 adds support for an unnamed generic address space. Pointers that are declared without pointing to a named address space point to the generic address space. Before referring to the region pointed to, the pointer must be associated with a named address space. Functions written with pointer arguments and return values which do not declare an address space are defined to point to the generic address space.

kernel function arguments declared to be a pointer or an array of a type must point to one of the named address spaces `__global`, `__local` or `__constant`.

The named address spaces are a subset of the generic address space except for the `constant` address space.

A pointer to address space A can only be assigned to a pointer to the same address space A or a pointer to the generic address space. Casting a pointer to address space A to a pointer to address space is illegal if A and B are named address spaces and A is not the same as B.

Examples:

```
private int f() { ... }          // should generate an error

local int *f() { ... }           // allowed

local int * private f() { ... }; // should generate an error.
```

The `__global`, `__constant`, `__local`, `__private`, `global`, `constant`, `local`, and `private` names are reserved for use as address space qualifiers and shall not be used otherwise. The `__generic` and `generic` names are reserved for future use.

> The size of pointers to different address spaces may differ. It is not correct to assume that, for example, `sizeof(__global int *)` always equals `sizeof(__local int *)`.

6.5.1. `__global` (or `global`)

The `__global` or `global` address space name is used to refer to memory objects (buffer or image objects) allocated from the `global` memory pool.

A buffer memory object can be declared as a pointer to a scalar, vector or user-defined struct. This allows the kernel to read and/or write any location in the buffer.

The actual size of the array memory object is determined when the memory object is allocated via appropriate API calls in the host code.

Some examples are:

```
global float4 *color; // An array of float4 elements

typedef struct {
    float a[3];
    int b[2];
} foo_t;

global foo_t *my_info; // An array of foo_t elements.
```

As image objects are always allocated from the `global` address space, the `__global` or `global` qualifier should not be specified for image types. The elements of an image object cannot be directly accessed. Built-in functions to read from and write to an image object are provided.

Variables defined at program scope and `static` variables inside a function can also be declared in the `global` address space. They can be defined with any valid OpenCL C data type except for those in Other Built-in Data Types. In particular, such program scope variables may be of any user-defined type, or a pointer to a user-defined type. In the presence of shared virtual memory, these pointers or pointer members should work as expected as long as they are shared virtual memory pointers and the referenced storage has been mapped appropriately. These variables in the `global` address space have the same lifetime as the program, and their values persist between calls to any of the kernels in the program. These variables are not shared across devices. They have distinct storage.

Program scope and `static` variables in the `global` address space may be initialized, but only with constant expressions.

Examples:

```
global int foo;         // OK.
int foo;                // OK. Declared in the global address space
global uchar buf[512];  // OK.
global int baz = 12;    // OK. Initialization is allowed
static global int bat;  // OK. Internal linkage

static int foo;         // OK. Declared in the global address space
static global int foo;  // OK.

int *foo;               // OK. foo is allocated in global address space.
                        // pointer to foo in generic address space

void func(...)
{
    int *foo;           // OK. foo is allocated in private address space.
                        // foo points to a location in generic address space.
    ...
}

global int * global ptr;          // OK.
int * global ptr;                 // OK.
constant int *global ptr=&baz;    // error since baz is in global address
                                  // space.
global int * constant ptr = &baz; // OK

// Pointers work. Also, initialization to a constant known at
// program load time
global int *global baz_ptr = &baz;

global image2d_t im; // Error. Invalid type for program scope
                     // variables

global event_t ev;   // Error. Invalid type for program scope variables

global int *bad_ptr; // Error. No implicit address space
```

The `const` qualifier can also be used with the `__global` qualifier to specify a read-only buffer memory object.

6.5.2. `__local` (or `local`)

The `__local` or `local` address space name is used to describe variables that need to be allocated in local memory and are shared by all work-items of a work-group. Pointers to the `__local` address space are allowed as arguments to functions (including kernel functions). Variables declared in the `__local` address space inside a kernel function must occur at kernel function scope.

Some examples of variables allocated in the `__local` address space inside a kernel function are:

```
kernel void my_func(...)
{
    local float a;     // A single float allocated
                       // in local address space

    local float b[10]; // An array of 10 floats
                       // allocated in local address space.

    if (...)
    {
        // example of variable in __local address space but not
        // declared at __kernel function scope.
        local float c; // not allowed.
    }
}
```

Variables allocated in the `__local` address space inside a kernel function cannot be initialized.

```
kernel void my_func(...)
{
    local float a = 1; // not allowed

    local float b;
    b = 1;             // allowed
}
```

Variables allocated in the `__local` address space inside a kernel function are allocated for each work-group executing the kernel and exist only for the lifetime of the work-group executing the kernel.

6.5.3. `__constant` (or `constant`)

The `__constant` or `constant` address space name is used to describe variables allocated in `global` memory and which are accessed inside a kernel(s) as read-only variables. These read-only variables can be accessed by all (global) work-items of the kernel during its execution. Pointers to the `__constant` address space are allowed as arguments to functions (including kernel functions) and for variables declared inside functions.

All string literal storage shall be in the `__constant` address space.

Each argument to a kernel that is a pointer to the `__constant` address space is counted separately towards the maximum number of such arguments, defined as the value of the `CL_DEVICE_MAX_CONSTANT_ARGS` device query.

Variables in the program scope can be declared in the `__constant` address space. Variables in the outermost scope of kernel functions can be declared in the `__constant` address space. These variables are required to be initialized and the values used to initialize these variables must be a compile time constant. Writing to such a variable results in a compile-time error.

Implementations are not required to aggregate these declarations into the fewest number of constant arguments. This behavior is implementation defined.

Thus portable code must conservatively assume that each variable declared inside a function or in program scope allocated in the `__constant` address space counts as a separate constant argument.

6.5.4. `__private` (or `private`)

Variables inside a kernel function not declared with an address space qualifier, all variables inside non-kernel functions, and all function arguments are in the `__private` or `private` address space. ~~**Variables declared as pointers are considered to point to the `__private` address space if an address space qualifier is not specified**~~.

6.5.5. The generic address space

The following rules apply when using pointers that point to the generic address space:

- A pointer that points to the `global`, `local` or `private` address space can be implicitly converted to a pointer to the unnamed generic address space but not vice-versa.
- Pointer casts can be used to cast a pointer that points to the `global`, `local` or `private` space to the unnamed generic address space and vice-versa.
- A pointer that points to the `constant` address space cannot be cast or implicitly converted to the generic address space.

A few examples follow.

This is the canonical example. In this example, function `foo` is declared with an argument that is a pointer with no address space qualifier.

```
void foo(int *a)
{
    *a = *a + 2;
}

kernel void k1(local int *a)
{
    ...
    foo(a);
    ...
}

kernel void k2(global int *a)
{
    ...
    foo(a);
    ...
}
```

In the example below, `var` is in the unnamed generic address space which gets mapped to the `global` or `local` address space depending on the result of the conditional expression.

```
kernel void bar(global int *g, local int *l)
{
    int *var;

    if (is_even(get_global_id(0))
        var = g;
    else
        var = l;
    *var = 42;
    ...
}
```

The example below is an example with one unnamed generic address space pointer with multiple named address space assignments.

```
int *ptr;
global int g;
ptr = &g; // legal

local int l;
ptr = &l; // legal

private int p;
ptr = &p; // legal

constant int c;
ptr = &c; // illegal
```

The example below is an example with one unnamed generic address space pointer being assigned to point to several named address spaces.

```
global int *gp;
local int *lp;
private int *pp;

int *p;
p = gp; // legal
p = lp; // legal
p = pp; // legal

// it is illegal to convert from a generic pointer
// to an explicit address space pointer without a cast:
gp = p; // compile-time error
lp = p; // compile-time error
pp = p; // compile-time error
```

6.5.6. Changes to C99

This section details the modifications to the C99 Specification needed to incorporate the functionality of named address space and the generic address space:

Clause 6.2.5 - Types, replace paragraph 26 with the following paragraphs:

If type `T` is qualified by the address space qualifier for address space `A`, then " `T` is in `A` ". If type `T` is in address space `A`, a pointer to `T` is also a " pointer into `A` " and the referenced address space of the pointer is `A`.

A pointer to `void` in any address space shall have the same representation and alignment requirements as a pointer to a character type in the same address space. Similarly, pointers to differently access-qualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types in the same address space shall have the same representation and alignment requirements as each other. All pointers to union types in the same address space shall have the same representation and alignment requirements as each other.

Clause 6.3.2.3 - Pointers, replace the first two paragraphs with the following paragraphs:

If a pointer into one address space is converted to a pointer into another address space, then unless the original pointer is a null pointer or the location referred to by the original pointer is within the second address space, the behavior is undefined. (For the original pointer to refer to a location within the second address space, the two address spaces must overlap).

A pointer to `void` in any address space may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type in some address space may be converted to a pointer to `void` in an enclosing address space and back again; the result shall compare equal to the original pointer.

For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q* -qualified version of the type (but with the same address-space qualifier or the generic address space); the values stored in the original and converted pointers shall compare equal.

Clause 6.3.2.3 - Pointers, replace the last sentence of paragraph 4 with:

Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers whose referenced address spaces overlap shall compare equal.

Clause 6.5.2.2 - Function calls, change the second bullet of paragraph 6 to:

both types are pointers to qualified or unqualified versions of a character type or `void` in the same address space or one type is a pointer in a named address space and the other is a pointer in the generic address space.

Clause 6.5.6 - Additive operators, add another constraint paragraph:

For subtraction, if the two operands are pointers into different address spaces, the address spaces must overlap.

Clause 6.5.8 - Relational operators, add another constraint paragraph:

If the two operands are pointers into different address spaces, the address spaces must overlap.

Clause 6.5.8 - Relational operators, add a new paragraph between existing paragraphs 3 and 4:

If the two operands are pointers into different address spaces, one of the address spaces encloses the other. The pointer into the enclosed address space is first converted to a pointer to the same reference type except with any address-space qualifier removed and any address-space qualifier of the other pointer's reference type added. (After this conversion, both pointers are pointers into the same address space).

Examples:

```
kernel void test1()
{
    global int arr[5] = { 0, 1, 2, 3, 4 };
    int *p = &arr[1];
    global int *q = &arr[3];

    // q implicitly converted to the generic address space
    // since the generic address space encloses the global
    // address space
    if (q >= p)
        printf("true\n");

    // q implicitly converted to the generic address space
    // since the generic address space encloses the global
    // address space
    if (p <= q)
        printf("true\n");
}
```

Clause 6.5.9 - Equality operators, add another constraint paragraph:

If the two operands are pointers into different address spaces, the address spaces must overlap.

Clause 6.5.9 - Equality operators, replace paragraph 5 with:

Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If both operands are pointers, each of the following conversions is performed as applicable:

- If the two operands are pointers into different address spaces, one of the address spaces encloses the other. The pointer into the enclosed address space is first converted to a pointer to the same reference type except with any address-space qualifier removed and any address-space qualifier of the other pointer's reference type added. (After this conversion, both pointers are pointers into the same address space).
- Then, if one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`, the former is converted to the type of the latter.

Examples:

```
int *ptr = NULL;
local int lval = SOME_VAL;
local int *lptr = &lval;
global int gval = SOME_OTHER_VAL;
global int *gptr = &gval;

ptr = lptr;

if (ptr == gptr) // legal
{
    ...
}

if (ptr == lptr) // legal
{
    ...
}

if (lptr == gptr) // illegal, compiler error
{
    ...
}
```

Consider the following example:

```
bool callee(int *p1, int *p2)
{
    if (p1 == p2)
        return true;
    return false;
}

void caller()
{
    global int *gptr = 0xdeadbeef;
    private int *pptr = 0xdeadbeef;

    // behavior of callee is undefined
    bool b = callee(gptr, pptr);
}
```

The behavior of callee is undefined as gptr and pptr are in different address spaces. The example above would have the same undefined behavior if the equality operator is replaced with a relational operator.

Examples:

```
int *ptr = NULL;
local int *lptr = NULL;
global int *gptr = NULL;

if (ptr == NULL) // legal
{
    ...
}

if (ptr == lptr) // legal
{
    ...
}

if (lptr == gptr) // compile-time error
{
    ...
}

ptr = lptr; // legal

intptr l = (intptr_t)lptr;
if (l == 0) // legal
{
    ...
}

if (l == NULL) // legal
{
    ...
}
```

Clause 6.5.9 - Equality operators, replace first sentence of paragraph 6 with:

Two pointers compare equal if and only if both are null pointers with overlapping address spaces.

Clause 6.5.15 - Conditional operator, add another constraint paragraph:

If the second and third operands are pointers into different address spaces, the address spaces must overlap.

Examples:

```
kernel void test1()
{
    global int arr[5] = { 0, 1, 2, 3, 4 };
    int *p = &arr[1];
    global int *q = &arr[3];
    local int *r = NULL;
    int *val = NULL;

    // legal. 2nd and 3rd operands are in address spaces
    // that overlap
    val = (q >= p) ? q : p;

    // compiler error. 2nd and 3rd operands are in disjoint
    // address spaces
    val = (q >= p) ? q : r;
}
```

Clause 6.5.16.1 - Simple assignment, change the third and fourth bullets of paragraph 1 to:

- both operands are pointers to qualified or unqualified versions of compatible types, the referenced address space of the left encloses the referenced address space of the right, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
- one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`, the referenced address space of the left encloses the referenced address space of the right, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.

Examples:

```
kernel void f()
{
int *ptr;
local int *lptr;
global int *gptr;
local int val = 55;

ptr = &val;  // legal: implicit cast to generic, then assign
lptr = ptr;  // illegal: no implicit cast from
             // generic to local
lptr = gptr; // illegal: no implicit cast from
             // global to local
ptr = gptr;  // legal: implicit cast from global to generic,
             // then assign
}
```

Clause 6.7.2.1 - Structure and union specifiers, add a new constraint paragraph:

Within a structure or union specifier, the type of a member shall not be qualified by an address space qualifier.

Clause 6.7.3 - Type qualifiers, add three new constraint paragraphs:

No type shall be qualified by qualifiers for two or more different address spaces.

6.6. Access Qualifiers

Image objects specified as arguments to a kernel can be declared to be read-only, write-only or read-write. ~~A kernel cannot read from and write to the same image object~~. The `__read_only` (or read_only) and `__write_only` (or `write_only`) qualifiers must be used with image object arguments to declare if the image object is being read or written by a kernel or function. The `__read_write` (or `read_write`) qualifier must be used with image object arguments of kernels and of user-defined functions to declare if the image object is being both read and written by the kernel. If no qualifier is provided, `read_only` is assumed.

In the following example

```
kernel void
foo (read_only image2d_t imageA,
     write_only image2d_t imageB)
{
    ...
}
```

imageA is a read-only 2D image object, and image is a write-only 2D image object.

The sampler-less read image and write image built-ins can be used with image declared with the `__read_write` (or `read_write`) qualifier. Calls to built-ins that read from an image using a sampler for images declared with the `__read_write` (or `read_write`) qualifier will be a compilation error.

Pipe objects specified as arguments to a kernel also use these access qualifiers. See the detailed description on how these access qualifiers can be used with pipes.

The `__read_only`, `__write_only`, `__read_write`, `read_only`, `write_only` and `read_write` names are reserved for use as access qualifiers and shall not be used otherwise.

6.7. Function Qualifiers

6.7.1. `__kernel` (or `kernel`)

The `__kernel` (or `kernel`) qualifier declares a function to be a kernel that can be executed by an application on an OpenCL device(s). The following rules apply to functions that are declared with this qualifier:

- It can be executed on the device only
- It can be called by the host

- It is just a regular function call if a `__kernel` function is called by another kernel function.

> Kernel functions with variables declared inside the function with the `__local` or `local` qualifier can be called by the host using appropriate APIs such as **clEnqueueNDRangeKernel**.

The `__kernel` and `kernel` names are reserved for use as functions qualifiers and shall not be used otherwise.

6.7.2. Optional Attribute Qualifiers

The `__kernel` qualifier can be used with the keyword *attribute* to declare additional information about the kernel function as described below.

The optional `__attribute__((vec_type_hint(<type>)))`[24] is a hint to the compiler and is intended to be a representation of the computational *width* of the `__kernel`, and should serve as the basis for calculating processor bandwidth utilization when the compiler is looking to autovectorize the code. In the `__attribute__((vec_type_hint(<type>)))` qualifier <type> is one of the built-in vector types listed in Built-in Vector Data Types or the constituent scalar element types. If `vec_type_hint (<type>)` is not specified, the kernel is assumed to have the `__attribute__((vec_type_hint(int)))` qualifier.

[24] Implicit in autovectorization is the assumption that any libraries called from the `__kernel` must be recompilable at run time to handle cases where the compiler decides to merge or separate workitems. This probably means that such libraries can never be hard coded binaries or that hard coded binaries must be accompanied either by source or some retargetable intermediate representation. This may be a code security question for some.

For example, where the developer specified a width of `float4`, the compiler should assume that the computation usually uses up to 4 lanes of a `float` vector, and would decide to merge work-items or possibly even separate one work-item into many threads to better match the hardware capabilities. A conforming implementation is not required to autovectorize code, but shall support the hint. A compiler may autovectorize, even if no hint is provided. If an implementation merges N work-items into one thread, it is responsible for correctly handling cases where the number of `global` or `local` work-items in any dimension modulo N is not zero.

Examples:

```
// autovectorize assuming float4 as the
// basic computation width
__kernel __attribute__((vec_type_hint(float4)))
void foo( __global float4 *p ) { ... }

// autovectorize assuming double as the
// basic computation width
__kernel __attribute__((vec_type_hint(double)))
void foo( __global float4 *p ) { ... }

// autovectorize assuming int (default)
// as the basic computation width
__kernel
void foo( __global float4 *p ) { ... }
```

If for example, a `__kernel` function is declared with

```
__attribute__(( vec_type_hint (float4)))
```

(meaning that most operations in the `__kernel` function are explicitly vectorized using `float4`) and the kernel is running using Intel® Advanced Vector Instructions (Intel® AVX) which implements a 8-float-wide vector unit, the autovectorizer might choose to merge two work-items to one thread, running a second work-item in the high half of the 256-bit AVX register.

As another example, a Power4 machine has two scalar double precision floating-point units with an 6-cycle deep pipe. An autovectorizer for the Power4 machine might choose to interleave six kernels declared with the `__attribute__(( vec_type_hint (double2)))` qualifier into one hardware thread, to ensure that there is always 12-way parallelism available to saturate the FPUs. It might also choose to merge 4 or 8 work-items (or some other number) if it concludes that these are better choices, due to resource utilization concerns or some preference for divisibility by 2.

The optional `__attribute__((work_group_size_hint(X, Y, Z)))` is a hint to the compiler and is intended to specify the work-group size that may be used i.e. value most likely to be specified by the *local_work_size* argument to **clEnqueueNDRangeKernel**. For example, the `__attribute__((work_group_size_hint(1, 1, 1)))` is a hint to the compiler that the kernel will most likely be executed with a work-group size of 1.

The optional `__attribute__((reqd_work_group_size(X, Y, Z)))` is the work-group size that must be used as the *local_work_size* argument to **clEnqueueNDRangeKernel**. This allows the compiler to optimize the generated code appropriately for this kernel.

If `Z` is one, the *work_dim* argument to **clEnqueueNDRangeKernel** can be 2 or 3. If `Y` and `Z` are one, the *work_dim* argument to **clEnqueueNDRangeKernel** can be 1, 2 or 3.

The optional `__attribute__((nosvm))` qualifier can be used with a pointer variable to inform the compiler that the pointer does not refer to a shared virtual memory region.

> `__attribute__((nosvm))` is deprecated, and the compiler can ignore it.

6.8. Storage-Class Specifiers

The `typedef`, `extern` and `static` storage-class specifiers are supported. The `auto` and `register` storage-class specifiers are not supported.

The `extern` storage-class specifier can only be used for functions (kernel and non-kernel functions) and `global` variables declared in program scope or variables declared inside a function (kernel and non-kernel functions). The `static` storage-class specifier can only be used for non-kernel functions, `global` variables declared in program scope and variables inside a function declared in the `global` or `constant` address space.

Examples:

```
extern constant float4 noise_table[256];
static constant float4 color_table[256];

extern kernel void my_foo(image2d_t img);
extern void my_bar(global float *a);

kernel void my_func(image2d_t img, global float *a)
{
    extern constant float4 a;
    static constant float4 b = (float4)(1.0f); // OK.
    static float c;  // Error: No implicit address space
    global int hurl; // Error: Must be static
    ...
    my_foo(img);
    ...
    my_bar(a);
    ...
    while (1)
    {
        static global int inside; // OK.
        ...
    }
    ...
}
```

6.9. Restrictions

> Items struckthrough are restrictions in a previous version of OpenCL C that are no longer present in OpenCL C 2.0.

a. The use of pointers is somewhat restricted. The following rules apply:
   - Arguments to kernel functions declared in a program that are pointers must be declared with the `__global`, `__constant` or `__local` qualifier.
   - A pointer declared with the `__constant` qualifier can only be assigned to a pointer declared

with the `__constant` qualifier respectively.

- Pointers to functions are not allowed.
- ~~Arguments to kernel functions in a program cannot be declared as a pointer to a pointer(s). Variables inside a function or arguments to non-kernel functions in a program can be declared as a pointer to a pointer(s).~~

b. An image type (`image2d_t`, `image3d_t`, `image2d_array_t`, `image1d_t`, `image1d_buffer_t` or `image1d_array_t`) can only be used as the type of a function argument. An image function argument cannot be modified. Elements of an image can only be accessed using the built-in image read and write functions.

An image type cannot be used to declare a variable, a structure or union field, an array of images, a pointer to an image, or the return type of a function. An image type cannot be used with the `__global`, `__private`, `__local` and `__constant` address space qualifiers. ~~The `image3d_t` type cannot be used with the `__write_only` access qualifier unless the **cl_khr_3d_image_writes** extension is enabled. An image type cannot be used with the `__read_write` access qualifer which is reserved for future use.~~

The sampler type (`sampler_t`) can only be used as the type of a function argument or a variable declared in the program scope or the outermost scope of a kernel function. The behavior of a sampler variable declared in a non-outermost scope of a kernel function is implementation-defined. A sampler argument or variable cannot be modified.

The sampler type cannot be used to declare a structure or union field, an array of samplers, a pointer to a sampler, or the return type of a function. The sampler type cannot be used with the `__local` and `__global` address space qualifiers.

c. Bit-field struct members are currently not supported.

d. Variable length arrays and structures with flexible (or unsized) arrays are not supported.

e. Variadic macros and functions with the exception of printf and enqueue_kernel are not supported.

f. If a list of parameters in a function declaration is empty, the function takes no arguments. This is due to the above restriction on variadic prototypes.

g. Unless defined in the OpenCL specification, the library functions, macros, types, and constants defined in the C99 standard headers `assert.h`, `ctype.h`, `complex.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `limits.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stdio.h`, `stdlib.h`, `string.h`, `tgmath.h`, `time.h`, `wchar.h` and `wctype.h` are not available and cannot be included by a program.

h. The `auto` and `register` storage-class specifiers are not supported.

i. ~~Predefined identifiers are not supported.~~

j. Recursion is not supported.

k. The return type of a kernel function must be `void`.

l. Arguments to kernel functions in a program cannot be declared with the built-in scalar types `bool`, `size_t`, `ptrdiff_t`, `intptr_t`, and `uintptr_t` or a struct and/or union that contain fields declared to be one of these built-in scalar types. The size in bytes of these types are implementation-defined and in addition can also be different for the OpenCL device and the host processor making it difficult to allocate buffer objects to be passed as arguments to a kernel declared as pointer to these types.

m. `half` is not supported as `half` can be used as a storage format[25] only and is not a data type on which floating-point arithmetic can be performed.

n. Whether or not irreducible control flow is illegal is implementation defined.

o. ~~Built-in types that are less than 32-bits in size, i.e. `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort`, and `half`, have the following restriction:~~

   - ~~Writes to a pointer (or arrays) of type `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort`, and `half` or to elements of a struct that are of type `char`, `uchar`, `char2`, `uchar2`, `short` and `ushort` are not supported. Refer to *section 9.9* for additional information.~~

     ~~The kernel example below shows what memory operations are not supported on built-in types less than 32-bits in size.~~

```
kernel void
do_proc (__global char *pA, short b,
         __global short *pB)
{
    char x[100];
    __private char *px = x;
    int id = (int)get_global_id(0);
    short f;

    f = pB[id] + b; // is allowed
    px[1] = pA[1]; // error. px cannot be written.
    pB[id] = b; // error. pB cannot be written
}
```

p. The type qualifiers `const`, `restrict` and `volatile` as defined by the C99 specification are supported. These qualifiers cannot be used with `image2d_t`, `image3d_t`, `image2d_array_t`, `image2d_depth_t`, `image2d_array_depth_t`, `image1d_t`, `image1d_buffer_t` and `image1d_array_t` types. Types other than pointer types shall not use the `restrict` qualifier.

q. The event type (`event_t`) cannot be used as the type of a kernel function argument. The event type cannot be used to declare a program scope variable. The event type cannot be used to declare a structure or union field. The event type cannot be used with the `__local`, `__constant` and `__global` address space qualifiers.

r. The `clk_event_t`, `ndrange_t` and `reserve_id_t` types cannot be used as arguments to kernel functions that get enqueued from the host. The `clk_event_t` and `reserve_id_t` types cannot be declared in program scope.

s. The values returned by applying the `sizeof` operator to the `queue_t`, `clk_event_t`, `ndrange_t` and `reserve_id_t` types is implementation-defined.

t. Kernels enqueued by the host must continue to have their arguments that are a pointer to a type declared to point to a named address space.

u. A function in an OpenCL program cannot be called `main`.

v. Implicit function declaration is not supported.

[25] Unless the **cl_khr_fp16** extension is supported and has been enabled.

6.10. Preprocessor Directives and Macros

The preprocessing directives defined by the C99 specification are supported.

The **#pragma** directive is described as:

> **#pragma** *pp-tokens*$_{opt}$ *new-line*

A **#pragma** directive where the preprocessing token `OPENCL` (used instead of **`STDC`**) does not immediately follow **pragma** in the directive (prior to any macro replacement) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **pragma** that is not recognized by the implementation is ignored. If the preprocessing token `OPENCL` does immediately follow **#pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms whose meanings are described elsewhere:

```
// on-off-switch is one of ON, OFF, or DEFAULT
#pragma OPENCL FP_CONTRACT on-off-switch

#pragma OPENCL EXTENSION extensionname : behavior

#pragma OPENCL EXTENSION all : behavior
```

The following predefined macro names are available.

`__FILE__`
: The presumed name of the current source file (a character string literal).

`__LINE__`
: The presumed line number (within the current source file) of the current source line (an integer constant).

`__OPENCL_VERSION__`
: Substitutes an integer reflecting the version number of the OpenCL supported by the OpenCL device. The version of OpenCL described in this document will have `__OPENCL_VERSION__` substitute the integer 200.

`CL_VERSION_1_0`
: Substitutes the integer 100 reflecting the OpenCL 1.0 version.

`CL_VERSION_1_1`
: Substitutes the integer 110 reflecting the OpenCL 1.1 version.

`CL_VERSION_1_2`

Substitutes the integer 120 reflecting the OpenCL 1.2 version.

`CL_VERSION_2_0`

Substitutes the integer 200 reflecting the OpenCL 2.0 version.

`__OPENCL_C_VERSION__`

Substitutes an integer reflecting the OpenCL C version specified by the `-cl-std` build option (see section 5.8.4.5 of the OpenCL Specification) to **clBuildProgram** or **clCompileProgram**. If the `-cl-std` build option is not specified, the highest OpenCL C 1.x language version supported by each device is used as the version of OpenCL C when compiling the program for each device. The version of OpenCL C described in this document will have `__OPENCL_C_VERSION__` substitute the integer 200 if `-cl-std=CL2.0` is specified.

`__ENDIAN_LITTLE__`

Used to determine if the OpenCL device is a little endian architecture or a big endian architecture (an integer constant of 1 if device is little endian and is undefined otherwise). Also refer to the value of the `CL_DEVICE_ENDIAN_LITTLE` device query.

`__kernel_exec(X, type__n__)` (and `kernel_exec(X, typen)`) is defined as

```
__kernel __attribute__((work_group_size_hint(X, 1, 1))) \
    __attribute__((vec_type_hint(type__n__)))
```

`__IMAGE_SUPPORT__`

Used to determine if the OpenCL device supports images. This is an integer constant of 1 if images are supported and is undefined otherwise. Also refer to the value of the `CL_DEVICE_IMAGE_SUPPORT` device query.

`__FAST_RELAXED_MATH__`

Used to determine if the `-cl-fast-relaxed-math` optimization option is specified in build options given to **clBuildProgram** or **clCompileProgram**. This is an integer constant of 1 if the `-cl-fast-relaxed-math` build option is specified and is undefined otherwise.

The `NULL` macro expands to a null pointer constant. An integer constant expression with the value 0, or such an expression cast to type `void *` is called a *null pointer constant*.

The macro names defined by the C99 specification but not currently supported by OpenCL are reserved for future use.

The predefined identifier `__func__` is available.

6.11. Attribute Qualifiers

This section describes the syntax with which `__attribute__` may be used, and the constructs to which attribute specifiers bind.

An attribute specifier is of the form

`__attribute__ ((_attribute-list_))`.

An attribute list is defined as:

attribute-list :

attribute$_{opt}$
attribute-list , *attribute*$_{opt}$

attribute :

attribute-token attribute-argument-clause$_{opt}$

attribute-token :

identifier

attribute-argument-clause :

(*attribute-argument-list*)

attribute-argument-list :

attribute-argument
attribute-argument-list , *attribute-argument*

attribute-argument :

assignment-expression

This syntax is taken directly from GCC but unlike GCC, which allows attributes to be applied only to functions, types, and variables, OpenCL attributes can be associated with:

- types;
- functions;
- variables;
- blocks; and
- control-flow statements.

In general, the rules for how an attribute binds, for a given context, are non-trivial and the reader is pointed to GCC's documentation and Maurer and Wong's paper [See 16. and 17. in *section 11 -* **References**] for the details.

6.11.1. Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of enum, struct and union types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Two attributes are currently defined for types: aligned, and packed.

You may specify type attributes in an enum, struct or union type declaration or definition, or for other types in a `typedef` declaration.

For an enum, struct or union type, you may specify attributes either between the enum, struct or union tag and the name of the type, or just past the closing curly brace of the *definition*. The former syntax is preferred.

`aligned (alignment)`

This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

force the compiler to insure (as far as it can) that each variable whose type is `struct S` or `more_aligned_int` will be allocated and aligned *at least* on a 8-byte boundary.

Note that the alignment of any given struct or union type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the struct or union in question and must also be a power of two. This means that you *can* effectively adjust the alignment of a struct or union type by attaching an aligned attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire struct or union type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given struct or union type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. In the example above, the size of each `short` is 2 bytes, and therefore the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that the effectiveness of aligned attributes may be limited by inherent limitations of the OpenCL device and compiler. For some devices, the OpenCL compiler may only be able to arrange for variables to be aligned up to a certain maximum alignment. If the OpenCL compiler is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well. See below.

`packed`

This attribute, attached to struct or union type definition, specifies that each member of the structure or union is placed to minimize the memory required. When attached to an enum definition, it indicates that the smallest integral type should be used.

Specifying this attribute for struct and union types is equivalent to specifying the packed attribute on each of the structure or union members.

In the following example struct `my_packed_struct’s members are packed closely together, but the internal layout of its `s` member is not packed. To do that, struct `my_unpacked_struct` would need to be packed, too.

```
struct my_unpacked_struct
{
    char c;
    int i;
};

struct __attribute__ ((packed)) my_packed_struct
{
    char c;
    int i;
    struct my_unpacked_struct s;
};
```

You may only specify this attribute on the definition of a enum, struct or union, not on a `typedef` which does not also define the enumerated type, structure or union.

6.11.2. Specifying Attributes of Functions

See Function Qualifiers for the function attribute qualifiers currently supported.

6.11.3. Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. The following attribute qualifiers are currently defined:

`aligned (alignment)`

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. The alignment value specified must be a power of two.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the OpenCL compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target device you are compiling for.

When used on a struct, or struct member, the aligned attribute can only increase the alignment; in order to decrease it, the packed attribute must be specified as well. When used as part of a `typedef`, the aligned attribute can both increase and decrease alignment, and specifying the packed attribute will generate a warning.

Note that the effectiveness of aligned attributes may be limited by inherent limitations of the OpenCL device and compiler. For some devices, the OpenCL compiler may only be able to arrange for variables to be aligned up to a certain maximum alignment. If the OpenCL compiler is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

`packed`

The packed attribute specifies that a variable or structure field should have the smallest possible alignment — one byte for a variable, unless you specify a larger value with the aligned attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows a:

```
struct foo
{
    char a;
    int x[2] __attribute__ ((packed));
};
```

An attribute list placed at the beginning of a user-defined type applies to the variable of that type and not the type, while attributes following the type body apply to the type.

For example:

```
/* a has alignment of 128 */
__attribute__((aligned(128))) struct A {int i;} a;

/* b has alignment of 16 */
__attribute__((aligned(16))) struct B {double d;}
__attribute__((aligned(32))) b ;

struct A a1; /* a1 has alignment of 4 */

struct B b1; /* b1 has alignment of 32 */
```

`endian (endiantype)`

The endian attribute determines the byte ordering of a variable. *endiantype* can be set to `host` indicating the variable uses the endianness of the host processor or can be set to `device` indicating the variable uses the endianness of the device on which the kernel will be executed. The default is `device`.

For example:

```
global float4 *p __attribute__ ((endian(host)));
```

specifies that data stored in memory pointed to by p will be in the host endian format.

The endian attribute can only be applied to pointer types that are in the `global` or `constant` address space. The endian attribute cannot be used for variables that are not a pointer type. The endian attribute value for both pointers must be the same when one pointer is assigned to another.

6.11.4. Specifying Attributes of Blocks and Control-Flow-Statements

For basic blocks and control-flow-statements the attribute is placed before the structure in question, for example:

```
__attribute__((attr1)) {...}

for __attribute__((attr2)) (...) __attribute__((attr3)) {...}
```

Here `attr1` applies to the block in braces and `attr2` and `attr3` apply to the loop's control construct and body, respectively.

No attribute qualifiers for blocks and control-flow-statements are currently defined.

6.11.5. Specifying Attribute For Unrolling Loops

The `__attribute__((opencl_unroll_hint))` and `__attribute__((opencl_unroll_hint(n)))` attribute qualifiers can be used to specify that a loop (for, while and do loops) can be unrolled. This attribute qualifier can be used to specify full unrolling or partial unrolling by a specified amount. This is a compiler hint and the compiler may ignore this directive.

n is the loop unrolling factor and must be a positive integral compile time constant expression. An unroll factor of 1 disables unrolling. If n is not specified, the compiler determines the unrolling factor for the loop.

> The `__attribute__((opencl_unroll_hint(n)))` attribute qualifier must appear immediately before the loop to be affected.

Examples:

```
__attribute__((opencl_unroll_hint(2)))
while (*s != 0)
    *p++ = *s++;
```

The tells the compiler to unroll the above while loop by a factor of 2.

```
__attribute__((opencl_unroll_hint))
for (int i=0; i<2; i++)
{
    ...
}
```

In the example above, the compiler will determine how much to unroll the loop.

```
__attribute__((opencl_unroll_hint(1)))
for (int i=0; i<32; i++)
{
    ...
}
```

The above is an example where the loop should not be unrolled.

Below are some examples of invalid usage of `__attribute__((opencl_unroll_hint(n)))`.

```
__attribute__((opencl_unroll_hint(-1)))
while (...)
{
    ...
}
```

The above example is an invalid usage of the loop unroll factor as the loop unroll factor is negative.

```
__attribute__((opencl_unroll_hint))
if (...)
{
    ...
}
```

The above example is invalid because the unroll attribute qualifier is used on a non-loop construct

```
kernel void
my_kernel( ... )
{
    int x;
    __attribute__((opencl_unroll_hint(x))
    for (int i=0; i<x; i++)
    {
        ...
    }
}
```

The above example is invalid because the loop unroll factor is not a compile-time constant expression.

6.11.6. Extending Attribute Qualifiers

The attribute syntax can be extended for standard language extensions and vendor specific extensions. Any extensions should follow the naming conventions outlined in the introduction to section 9 in the OpenCL 2.0 Extension Specification.

Attributes are intended as useful hints to the compiler. It is our intention that a particular implementation of OpenCL be free to ignore all attributes and the resulting executable binary will produce the same result. This does not preclude an implementation from making use of the additional information provided by attributes and performing optimizations or other transformations as it sees fit. In this case it is the programmer's responsibility to guarantee that the information provided is in some sense correct.

6.12. Blocks

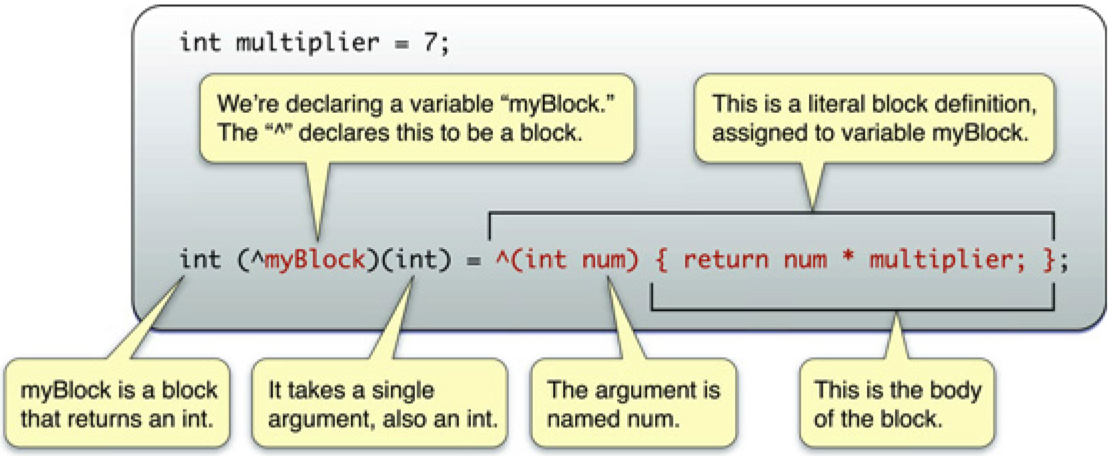
OpenCL C 2.0 adds support for the clang block syntax[26]. Like function types, the Block type is a pair consisting of a result value type and a list of parameter types very similar to a function type. Blocks are intended to be used much like functions with the key distinction being that in addition to executable code they also contain various variable bindings to automatic (stack) or `global` memory.

[26] This syntax is already part of the clang source tree on which most vendors have based their OpenCL implementations. Additionally, blocks based closures are supported by the clang open source C compiler as well as Mac OS X's C and Objective C compilers. Specifically, Mac OS X's Grand Central Dispatch allows applications to queue tasks as a block.

6.12.1. Declaring and Using a Block

You use the ^ operator to declare a Block variable and to indicate the beginning of a Block literal. The body of the Block itself is contained within {}, as shown in this example (as usual with C, ; indicates the end of the statement):

The example is explained in the following illustration:



Notice that the Block is able to make use of variables from the same scope in which it was defined.

If you declare a Block as a variable, you can then use it just as you would a function:

```
int multiplier = 7;

int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};

printf("%d\n", myBlock(3));
// prints 21
```

6.12.2. Declaring a Block Reference

Block variables hold references to Blocks. You declare them using syntax similar to that you use to declare a pointer to a function, except that you use ^ instead of *. The Block type fully interoperates with the rest of the C type system. The following are valid Block variable declarations:

```
void (^blockReturningVoidWithVoidArgument)(void);
int (^blockReturningIntWithIntAndCharArguments)(int, char);
```

A Block that takes no arguments must specify `void` in the argument list. A Block reference may not be dereferenced via the pointer dereference operation *, and thus a Block's size may not be computed at compile time.

Blocks are designed to be fully type safe by giving the compiler a full set of metadata to use to validate use of Blocks, parameters passed to blocks, and assignment of the return value.

You can also create types for Blocks — doing so is generally considered to be best practice when you use a block with a given signature in multiple places:

```
typedef float (^MyBlockType)(float, float);

MyBlockType myFirstBlock = // ...;
MyBlockType mySecondBlock = // ...;
```

6.12.3. Block Literal Expressions

A Block literal expression produces a reference to a Block. It is introduced by the use of the ^ token as a unary operator.

Block_literal_expression :
  ^ *block_decl compound_statement_body*

block_decl :
  empty
  parameter_list
  type_expression

where *type_expression* is extended to allow ^ as a Block reference where * is allowed as a function reference.

The following Block literal:

```
^ void (void) { printf("hello world**\n**"); }
```

produces a reference to a Block with no arguments with no return value.

The return type is optional and is inferred from the return statements. If the return statements return a value, they all must return a value of the same type. If there is no value returned the inferred type of the Block is `void`; otherwise it is the type of the return statement value. If the return type is omitted and the argument list is `( void )`, the `( void )` argument list may also be omitted.

So:

```
^ ( void ) { printf("hello world**\n**"); }
```

and:

```
^ { printf("hello world**\n**"); }
```

are exactly equivalent constructs for the same expression.

The compound statement body establishes a new lexical scope within that of its parent. Variables used within the scope of the compound statement are bound to the Block in the normal manner with the exception of those in automatic (stack) storage. Thus one may access functions and global variables as one would expect, as well as `static` local variables.

Local automatic (stack) variables referenced within the compound statement of a Block are imported and captured by the Block as const copies. The capture (binding) is performed at the time of the Block literal expression evaluation.

The compiler is not required to capture a variable if it can prove that no references to the variable will actually be evaluated.

The lifetime of variables declared in a Block is that of a function..

Block literal expressions may occur within Block literal expressions (nested) and all variables captured by any nested blocks are implicitly also captured in the scopes of their enclosing Blocks.

A Block literal expression may be used as the initialization value for Block variables at global or local `static` scope.

You can also declare a Block as a global literal in program scope.

```
int GlobalInt = 0;

int (^getGlobalInt)(void) = ^{ return GlobalInt; };
```

6.12.4. Control Flow

The compound statement of a Block is treated much like a function body with respect to control flow in that continue, break and goto do not escape the Block.

6.12.5. Restrictions

The following Blocks features are currently not supported in OpenCL C.

- The `__block` storage type.
- The **Block_copy**() and **Block_release**() functions that copy and release Blocks.
- Blocks with variadic arguments.
- Arrays of Blocks.
- Blocks as structures and union members.

Block literals are assumed to allocate memory at the point of definition and to be destroyed at the end of the same scope. To support these behaviors, additional restrictions[27] in addition to the above feature restrictions are:

[27] OpenCL C does not allow function pointers primarily because it is difficult or expensive to implement generic indirections to executable code in many hardware architectures that OpenCL targets. OpenCL C's design of Blocks is intended to respect that same condition, yielding the restrictions listed here. As such, Blocks allow a form of dynamically enqueued function scheduling

without providing a form of runtime synchronous dynamic dispatch analogous to function pointers.

- Block variables must be defined and used in a way that allows them to be statically determinable at build or “link to executable” time. In particular:
  - Block variables assigned in one scope must be used only with the same or any nested scope.
  - The `extern` storage-class specified cannot be used with program scope block variables.
  - Block variable declarations are implicitly qualified with const. Therefore all block variables must be initialized at declaration time and may not be reassigned.
  - A block cannot be the return value of a function.
- The unary operators (***** and **&**) cannot be used with a Block.
- Pointers to Blocks are not allowed.
- Blocks cannot be used as expressions of the ternary selection operator (**?:**).
- A Block cannot capture another Block variable declared in the outer scope (Example 4).
- Block capture semantics follows regular C argument passing convention, i.e. arrays are captured by reference (decayed to pointers) and structs are captured by value (Example 5).

Some examples that describe legal and illegal issue of Blocks in OpenCL C are described below.

Example 1:

```
void foo(int *x, int (^bar)(int, int))
{
    *x = bar(*x, *x);
}

kernel
void k(global int *x, global int *z)
{
    if (some expression)
        foo(x, ^int(int x, int y){return x+y+*z;}); // legal
    else
        foo(x, ^int(int x, int y){return (x*y)-*z;}); // legal
}
```

Example 2:

```
kernel
void k(global int *x, global int *z)
{
    int ^(tmp)(int, int);
    if (some expression)
    {
        tmp = ^int(int x, int y){return x+y+*z;}); // illegal
    }
    *x = foo(x, tmp);
}
```

Example 3:

```
int GlobalInt = 0;
int (^getGlobalInt)(void) = ^{ return GlobalInt; }; // legal
int (^getAnotherGlobalInt)(void);                   // illegal
extern int (^getExternGlobalInt)(void);             // illegal

void foo()

{
    ...
    getGlobalInt = ^{ return 0; }; // illegal - cannot assign to
                                   // a global block variable
    ...
}
```

Example 4:

```
void (^bl0)(void) = ^{
    ...
};

kernel void k()
{
    void(^bl1)(void) = ^{
        ...
    };

    void(^bl2)(void) = ^{
        bl0(); // legal because bl0 is a global
               // variable available in this scope
        bl1(); // illegal because bl1 would have to be captured
    };
}
```

Example 5:

```
struct v {
    int arr[2];
} s = {0, 1};

void (^bl1)() = ^(){printf("%d\n", s.arr[1]);};
// array content copied into captured struct location

int arr[2] = {0, 1};
void (^bl2)() = ^(){printf("%d\n", arr[1]);};
// array decayed to pointer while captured

s.arr[1] = arr[1] = 8;

bl1(); // prints - 1
bl2(); // prints - 8
```

6.13. Built-in Functions

The OpenCL C programming language provides a rich set of built-in functions for scalar and vector operations. Many of these functions are similar to the function names provided in common C libraries but they support scalar and vector argument types. Applications should use the built-in functions wherever possible instead of writing their own version.

User defined OpenCL C functions behave per C standard rules for functions as defined in section 6.9.1 of the C99 Specification. On entry to the function, the size of each variably modified parameter is evaluated and the value of each argument expression is converted to the type of the corresponding parameter as per the usual arithmetic conversion rules. Built-in functions described in this section behave similarly, except that in order to avoid ambiguity between multiple forms of the same built-in function, implicit scalar widening shall not occur. Note that some built-in functions described in this section do have forms that operate on mixed scalar and vector types, however.

6.13.1. Work-Item Functions

The following table describes the list of built-in work-item functions that can be used to query the number of dimensions, the global and local work size specified to **clEnqueueNDRangeKernel**, and the global and local identifier of each work-item when this kernel is being executed on a device.

Table 7. Work-Item Functions Table

Function	**Description**
uint **get_work_dim**()	Returns the number of dimensions in use. This is the value given to the *work_dim* argument specified in **clEnqueueNDRangeKernel**.

size_t **get_global_size**(uint *dimindx*)	Returns the number of global work-items specified for dimension identified by *dimindx*. This value is given by the *global_work_size* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_global_size**() returns 1.
size_t **get_global_id**(uint *dimindx*)	Returns the unique global work-item ID value for dimension identified by *dimindx*. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_global_id**() returns 0.
size_t **get_local_size**(uint *dimindx*)	Returns the number of local work-items specified in dimension identified by *dimindx*. This value is at most the value given by the *local_work_size* argument to **clEnqueueNDRangeKernel** if *local_work_size* is not `NULL`; otherwise the OpenCL implementation chooses an appropriate *local_work_size* value which is returned by this function. If the kernel is executed with a non-uniform work-group size[28], calls to this built-in from some work-groups may return different values than calls to this built-in from other work-groups. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_local_size**() returns 1.

size_t **get_enqueued_local_size**(uint *dimindx*)	Returns the same value as that returned by **get_local_size**(*dimindx*) if the kernel is executed with a uniform work-group size. If the kernel is executed with a non-uniform work-group size, returns the number of local work-items in each of the work-groups that make up the uniform region of the global range in the dimension identified by *dimindx*. If the *local_work_size* argument to **clEnqueueNDRangeKernel** is not `NULL`, this value will match the value specified in *local_work_size*[*dimindx*]. If *local_work_size* is `NULL`, this value will match the local size that the implementation determined would be most efficient at implementing the uniform region of the global range. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_enqueued_local_size**() returns 1.
size_t **get_local_id**(uint *dimindx*)	Returns the unique local work-item ID, i.e. a work-item within a specific work-group for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_local_id**() returns 0.
size_t **get_num_groups**(uint *dimindx*)	Returns the number of work-groups that will execute a kernel for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_num_groups**() returns 1.
size_t **get_group_id**(uint *dimindx*)	**get_group_id** returns the work-group ID which is a number from 0 .. **get_num_groups**(*dimindx*) - 1. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values, **get_group_id**() returns 0.
size_t **get_global_offset**(uint *dimindx*)	**get_global_offset** returns the offset values specified in *global_work_offset* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values, **get_global_offset**() returns 0.

size_t **get_global_linear_id**()	Returns the work-items 1-dimensional global ID. For 1D work-groups, it is computed as **get_global_id**(0) - **get_global_offset**(0). For 2D work-groups, it is computed as (**get_global_id**(1) - **get_global_offset**(1)) * **get_global_size**(0) + (**get_global_id**(0) - **get_global_offset**(0)). For 3D work-groups, it is computed as **get_global_id**(2) - **get_global_offset**(2 * **get_global_size**(1) * **get_global_size**(0)) + **get_global_id**(1) - **get_global_offset**(1 * **get_global_size** (0)) + (**get_global_id**(0) - **get_global_offset**(0)).
size_t **get_local_linear_id**()	Returns the work-items 1-dimensional local ID. For 1D work-groups, it is the same value as **get_local_id**(0). For 2D work-groups, it is computed as **get_local_id**(1) * **get_local_size**(0) + **get_local_id**(0). For 3D work-groups, it is computed as (**get_local_id**(2) * **get_local_size**(1) * **get_local_size**(0)) + (**get_local_id**(1) * **get_local_size**(0)) + **get_local_id**(0).

[28] I.e. the *global_work_size* values specified to **clEnqueueNDRangeKernel** are not evenly divisable by the *local_work_size* values for each dimension.

6.13.2. Math Functions

The built-in math functions are categorized into the following:

- A list of built-in functions that have scalar or vector argument versions, and,
- A list of built-in functions that only take scalar `float` arguments.

The vector versions of the math functions operate component-wise. The description is per-component.

The built-in math functions are not affected by the prevailing rounding mode in the calling environment, and always return the same value as they would if called with the round to nearest even rounding mode.

The following table describes the list of built-in math functions that can take scalar or vector arguments. We use the generic type name `gentype` to indicate that the function can take `float`, `float2`, `float3`, `float4`, `float8`, `float16`, `double`, `double2`, `double3`, `double4`, `double8` or `double16` as the type for the arguments. We use the generic type name `gentypef` to indicate that the function can take `float`, `float2`, `float3`, `float4`, `float8`, or `float16` as the type for the arguments. We use the generic type name `gentyped` to indicate that the function can take `double`, `double2`, `double3`, `double4`, `double8` or `double16` as the type for the arguments. For any specific use of a function, the actual type has to be the same for all arguments and the return type, unless otherwise specified.

Table 8. Scalar and Vector Argument Built-in Math Function Table

Function	Description
gentype **acos**(gentype)	Arc cosine function. Returns an angle in radians.
gentype **acosh**(gentype)	Inverse hyperbolic cosine. Returns an angle in radians.
gentype **acospi**(gentype *x*)	Compute **acos**(*x*) / π.
gentype **asin**(gentype)	Arc sine function. Returns an angle in radians.
gentype **asinh**(gentype)	Inverse hyperbolic sine. Returns an angle in radians.
gentype **asinpi**(gentype *x*)	Compute **asin**(*x*) / π.
gentype **atan**(gentype *y_over_x*)	Arc tangent function. Returns an angle in radians.
gentype **atan2**(gentype *y*, gentype *x*)	Arc tangent of *y* / *x*. Returns an angle in radians.
gentype **atanh**(gentype)	Hyperbolic arc tangent. Returns an angle in radians.
gentype **atanpi**(gentype *x*)	Compute **atan**(*x*) / π.
gentype **atan2pi**(gentype *y*, gentype *x*)	Compute **atan2**(*y*, *x*) / π.
gentype **cbrt**(gentype)	Compute cube-root.
gentype **ceil**(gentype)	Round to integral value using the round to positive infinity rounding mode.
gentype **copysign**(gentype *x*, gentype *y*)	Returns *x* with its sign changed to match the sign of *y*.
gentype **cos**(gentype *x*)	Compute cosine, where *x* is an angle in radians.
gentype **cosh**(gentype *x*)	Compute hyperbolic consine, where *x* is an angle in radians.
gentype **cospi**(gentype *x*)	Compute **cos**(π *x*).
gentype **erfc**(gentype)	Complementary error function.
gentype **erf**(gentype)	Error function encountered in integrating the *normal distribution*.
gentype **exp**(gentype *x*)	Compute the base-*e* exponential of *x*.
gentype **exp2**(gentype)	Exponential base 2 function.
gentype **exp10**(gentype)	Exponential base 10 function.
gentype **expm1**(gentype *x*)	Compute e^x - 1.0.

gentype **fabs**(gentype)	Compute absolute value of a floating-point number.
gentype **fdim**(gentype *x*, gentype *y*)	*x* - *y* if *x* > *y*, +0 if *x* is less than or equal to y.
gentype **floor**(gentype)	Round to integral value using the round to negative infinity rounding mode.
gentype **fma**(gentype *a*, gentype *b*, gentype *c*)	Returns the correctly rounded floating-point representation of the sum of *c* with the infinitely precise product of *a* and *b*. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
gentype **fmax**(gentype *x*, gentype *y*) gentypef **fmax**(gentypef *x*, float *y*) gentyped **fmax**(gentyped *x*, double *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*. If one argument is a NaN, **fmax**() returns the other argument. If both arguments are NaNs, **fmax**() returns a NaN.
gentype **fmin**[29](gentype *x*, gentype *y*) gentypef **fmin**(gentypef *x*, float *y*) gentyped **fmin**(gentyped *x*, double *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*. If one argument is a NaN, **fmin**() returns the other argument. If both arguments are NaNs, **fmin**() returns a NaN.
gentype **fmod**(gentype *x*, gentype *y*)	Modulus. Returns *x* - *y* * **trunc**(*x*/*y*).
gentype **fract**(gentype *x*, gentype **iptr*)[30]	Returns **fmin**(*x* - **floor**(*x*), `0x1.fffffep-1f`). **floor**(x) is returned in *iptr*.
float*n* **frexp**(float*n* *x*, int*n* *exp) float **frexp**(float *x*, int *exp)	Extract mantissa and exponent from *x*. For each component the mantissa returned is a `float` with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.
double*n* **frexp**(double*n* *x*, int*n* *exp) double **frexp**(double *x*, int *exp)	Extract mantissa and exponent from *x*. For each component the mantissa returned is a `float` with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.
gentype **hypot**(gentype *x*, gentype *y*)	Compute the value of the square root of $x^2 + y^2$ without undue overflow or underflow.
int*n* **ilogb**(float*n* *x*) int **ilogb**(float *x*) int*n* **ilogb**(double*n* *x*) int **ilogb**(double *x*)	Return the exponent as an integer value.
float*n* **ldexp**(float*n* *x*, int*n* *k*) float*n* **ldexp**(float*n* *x*, int *k*) float **ldexp**(float *x*, int *k*) double*n* **ldexp**(double*n* *x*, int*n* *k*) double*n* **ldexp**(double*n* *x*, int *k*) double **ldexp**(double *x*, int *k*)	Multiply *x* by 2 to the power *k*.
gentype **lgamma**(gentype *x*) float*n* **lgamma_r**(float*n* *x*, int*n* **signp*) float **lgamma_r**(float *x*, int **signp*) double*n* **lgamma_r**(double*n* *x*, int*n* **signp*) double **lgamma_r**(double *x*, int **signp*)	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the *signp* argument of **lgamma_r**.

gentype **log**(gentype)	Compute natural logarithm.
gentype **log2**(gentype)	Compute a base 2 logarithm.
gentype **log10**(gentype)	Compute a base 10 logarithm.
gentype **log1p**(gentype *x*)	Compute $\log_e(1.0 + x)$.
gentype **logb**(gentype *x*)	Compute the exponent of *x*, which is the integral part of $\log_r(\lvert x \rvert)$.
gentype **mad**(gentype *a*, gentype *b*, gentype *c*)	**mad** computes $a * b + c$. The function may compute $a * b + c$ with reduced accuracy in the embedded profile. See the SPIR-V OpenCL environment specification for details. On some hardware the mad instruction may provide better performance than expanded computation of $a * b + c$.[31]
gentype **maxmag**(gentype *x*, gentype *y*)	Returns *x* if $\lvert x \rvert > \lvert y \rvert$, *y* if $\lvert y \rvert > \lvert x \rvert$, otherwise **fmax**(*x*, *y*).
gentype **minmag**(gentype *x*, gentype *y*)	Returns *x* if $\lvert x \rvert < \lvert y \rvert$, *y* if $\lvert y \rvert < \lvert x \rvert$, otherwise **fmin**(*x*, *y*).
gentype **modf**(gentype *x*, gentype **iptr*)	Decompose a floating-point number. The **modf** function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by *iptr*.
float*n* **nan**(uint*n* *nancode*) float **nan**(uint *nancode*) double*n* **nan**(ulong*n* *nancode*) double **nan**(ulong *nancode*)	Returns a quiet NaN. The *nancode* may be placed in the significand of the resulting NaN.
gentype **nextafter**(gentype *x*, gentype *y*)	Computes the next representable single-precision floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, **nextafter**() returns the largest representable floating-point number less than *x*.
gentype **pow**(gentype *x*, gentype *y*)	Compute *x* to the power *y*.
float*n* **pown**(float*n* *x*, int*n* *y*) float **pown**(float *x*, int *y*) double*n* **pown**(double*n* *x*, int*n* *y*) double **pown**(double *x*, int *y*)	Compute *x* to the power *y*, where *y* is an integer.
gentype **powr**(gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0.
gentype **remainder**(gentype *x*, gentype *y*)	Compute the value *r* such that $r = x - n*y$, where *n* is the integer nearest the exact value of x/y. If there are two integers closest to x/y, *n* shall be the even one. If *r* is zero, it is given the same sign as *x*.

float*n* **remquo**(float*n* *x*, float*n* *y*, int*n* **quo*) float **remquo**(float *x*, float *y*, int **quo*)	The **remquo** function computes the value r such that *r* = *x* - *k***y*, where *k* is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *k* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.
double*n* **remquo**(double*n* *x*, double*n* *y*, int*n* * *quo*) double **remquo**(double *x*, double *y*, int **quo*)	The **remquo** function computes the value r such that *r* = *x* - *k***y*, where *k* is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *k* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.
gentype **rint**(gentype)	Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 for description of rounding modes.
float*n* **rootn**(float*n* *x*, int*n* *y*) float **rootn**(float *x*, int *y*) double*n* **rootn**(double*n* *x*, int*n* *y*) double*n* **rootn**(double *x*, int *y*)	Compute *x* to the power 1/*y*.
gentype **round**(gentype *x*)	Return the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction.
gentype **rsqrt**(gentype)	Compute inverse square root.
gentype **sin**(gentype *x*)	Compute sine, where *x* is an angle in radians.
gentype **sincos**(gentype *x*, gentype **cosval*)	Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in *cosval*, where *x* is an angle in radians.
gentype **sinh**(gentype *x*)	Compute hyperbolic sine, where *x* is an angle in radians
gentype **sinpi**(gentype *x*)	Compute **sin**(π *x*).
gentype **sqrt**(gentype)	Compute square root.
gentype **tan**(gentype *x*)	Compute tangent, where *x* is an angle in radians.
gentype **tanh**(gentype *x*)	Compute hyperbolic tangent, where *x* is an angle in radians.
gentype **tanpi**(gentype *x*)	Compute **tan**(π *x*).
gentype **tgamma**(gentype)	Compute the gamma function.

gentype **trunc**(gentype)	Round to integral value using the round to zero rounding mode.

[29] **fmin** and **fmax** behave as defined by C99 and may not match the IEEE 754-2008 definition for **minNum** and **maxNum** with regard to signaling NaNs. Specifically, signaling NaNs may behave as quiet NaNs.

[30] The **min**() operator is there to prevent **fract**(-small) from returning 1.0. It returns the largest positive floating-point number less than 1.0.

[31] The user is cautioned that for some usages, e.g. **mad**(a, b, -a*b), the definition of **mad**() is loose enough in the embedded profile that almost any result is allowed from **mad**() for some values of a and b.

The following table describes the following functions:

- A subset of functions from Scalar and Vector Argument Built-in Math Function Table that are defined with the half_ prefix . These functions are implemented with a minimum of 10-bits of accuracy, i.e. an ULP value <= 8192 ulp.
- A subset of functions from Scalar and Vector Argument Built-in Math Function Table that are defined with the native_ prefix. These functions may map to one or more native device instructions and will typically have better performance compared to the corresponding functions (without the `native_` prefix) described in Scalar and Vector Argument Built-in Math Function Table. The accuracy (and in some cases the input range(s)) of these functions is implementation-defined.
- `half_` and `native_` functions for following basic operations: divide and reciprocal.

We use the generic type name `gentype` to indicate that the functions in the following table can take `float`, `float2`, `float3`, `float4`, `float8` or `float16` as the type for the arguments.

Table 9. Scalar and Vector Argument Built-in half and native Math Functions

Function	**Description**
gentype **half_cos**(gentype *x*)	Compute cosine. *x* is an angle in radians, and must be in the range [-2^{16}, $+2^{16}$].
gentype **half_divide**(gentype *x*, gentype *y*)	Compute *x* / *y*.
gentype **half_exp**(gentype *x*)	Compute the base-*e* exponential of *x*.
gentype **half_exp2**(gentype *x*)	Compute the base- 2 exponential of *x*.
gentype **half_exp10**(gentype *x*)	Compute the base- 10 exponential of *x*.
gentype **half_log**(gentype *x*)	Compute natural logarithm.
gentype **half_log2**(gentype *x*)	Compute a base 2 logarithm.
gentype **half_log10**(gentype *x*)	Compute a base 10 logarithm.
gentype **half_powr**(gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0.
gentype **half_recip**(gentype *x*)	Compute reciprocal.
gentype **half_rsqrt**(gentype *x*)	Compute inverse square root.

gentype **half_sin**(gentype *x*)	Compute sine. *x* is an angle in radians, and must be in the range [-2^{16}, $+2^{16}$].
gentype **half_sqrt**(gentype *x*)	Compute square root.
gentype **half_tan**(gentype *x*)	Compute tangent. *x* is an angle in radians, and must be in the range [-2^{16}, $+2^{16}$].
gentype **native_cos**(gentype *x*)	Compute cosine over an implementation-defined range, where *x* is an angle in radians. The maximum error is implementation-defined.
gentype **native_divide**(gentype *x*, gentype *y*)	Compute *x* / *y* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp**(gentype *x*)	Compute the base-*e* exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp2**(gentype *x*)	Compute the base-2 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp10**(gentype *x*)	Compute the base-10 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log**(gentype *x*)	Compute natural logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log2**(gentype *x*)	Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log10**(gentype *x*)	Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_powr**(gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0. The range of *x* and *y* are implementation-defined. The maximum error is implementation-defined.
gentype **native_recip**(gentype *x*)	Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_rsqrt**(gentype *x*)	Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_sin**(gentype *x*)	Compute sine over an implementation-defined range, where *x* is an angle in radians. The maximum error is implementation-defined.
gentype **native_sqrt**(gentype *x*)	Compute square root over an implementation-defined range. The maximum error is implementation-defined.

gentype **native_tan**(gentype *x*)	Compute tangent over an implementation-defined range, where *x* is an angle in radians. The maximum error is implementation-defined.

Support for denormal values is optional for **half_** functions. The **half_** functions may return any result allowed by Edge Case Behavior, even when `-cl-denorms-are-zero` (see section 5.8.4.2 of the OpenCL Specification) is not in force. Support for denormal values is implementation-defined for **native_** functions.

The following symbolic constants are available. Their values are of type `float` and are accurate within the precision of a single precision floating-point number.

Constant Name	Description
`MAXFLOAT`	Value of maximum non-infinite single-precision floating-point number.
`HUGE_VALF`	A positive `float` constant expression. `HUGE_VALF` evaluates to +infinity. Used as an error value returned by the built-in math functions.
`INFINITY`	A constant expression of type `float` representing positive or unsigned infinity.
`NAN`	A constant expression of type `float` representing a quiet NaN.

If double precision is supported by the device, the following symbolic constants will also be available:

Constant Name	Description
`HUGE_VAL`	A positive double constant expression. `HUGE_VAL` evaluates to +infinity. Used as an error value returned by the built-in math functions.

Floating-point macros and pragmas

The `FP_CONTRACT` pragma can be used to allow (if the state is on) or disallow (if the state is off) the implementation to contract expressions. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined.

The pragma definition to set `FP_CONTRACT` is:

```
// on-off-switch is one of ON, OFF, or DEFAULT.
// The DEFAULT value is ON.
#pragma OPENCL FP_CONTRACT on-off-switch
```

The `FP_FAST_FMAF` macro indicates whether the **fma** function is fast compared with direct code for single precision floating-point. If defined, the `FP_FAST_FMAF` macro shall indicate that the **fma** function generally executes about as fast as, or faster than, a multiply and an add of `float` operands.

The macro names given in the following list must use the values specified. These constant expressions are suitable for use in `#if` preprocessing directives.

```
#define FLT_DIG          6
#define FLT_MANT_DIG     24
#define FLT_MAX_10_EXP   +38
#define FLT_MAX_EXP      +128
#define FLT_MIN_10_EXP   -37
#define FLT_MIN_EXP      -125
#define FLT_RADIX        2
#define FLT_MAX          0x1.fffffep127f
#define FLT_MIN          0x1.0p-126f
#define FLT_EPSILON      0x1.0p-23f
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
`FLT_DIG`	`CL_FLT_DIG`
`FLT_MANT_DIG`	`CL_FLT_MANT_DIG`
`FLT_MAX_10_EXP`	`CL_FLT_MAX_10_EXP`
`FLT_MAX_EXP`	`CL_FLT_MAX_EXP`
`FLT_MIN_10_EXP`	`CL_FLT_MIN_10_EXP`
`FLT_MIN_EXP`	`CL_FLT_MIN_EXP`
`FLT_RADIX`	`CL_FLT_RADIX`
`FLT_MAX`	`CL_FLT_MAX`
`FLT_MIN`	`CL_FLT_MIN`
`FLT_EPSILSON`	`CL_FLT_EPSILON`

The following macros shall expand to integer constant expressions whose values are returned by **ilogb**(*x*) if *x* is zero or NaN, respectively. The value of `FP_ILOGB0` shall be either `INT_MIN` or `-INT_MAX`. The value of `FP_ILOGBNAN` shall be either `INT_MAX` or `INT_MIN`.

The following constants are also available. They are of type `float` and are accurate within the precision of the `float` type.

Constant	Description

M_E_F	Value of *e*
M_LOG2E_F	Value of $\log_2 e$
M_LOG10E_F	Value of $\log_{10} e$
M_LN2_F	Value of $\log_e 2$
M_LN10_F	Value of $\log_e 10$
M_PI_F	Value of π
M_PI_2_F	Value of π / 2
M_PI_4_F	Value of π / 4
M_1_PI_F	Value of 1 / π
M_2_PI_F	Value of 2 / π
M_2_SQRTPI_F	Value of 2 / √π
M_SQRT2_F	Value of √2
M_SQRT1_2_F	Value of 1 / √2

If double precision is supported by the device, the following macros and constants are also available:

The FP_FAST_FMA macro indicates whether the **fma**() family of functions are fast compared with direct code for double precision floating-point. If defined, the FP_FAST_FMA macro shall indicate that the **fma**() function generally executes about as fast as, or faster than, a multiply and an add of double operands

The macro names given in the following list must use the values specified. These constant expressions are suitable for use in #if preprocessing directives.

```
#define DBL_DIG         15
#define DBL_MANT_DIG    53
#define DBL_MAX_10_EXP  +308
#define DBL_MAX_EXP     +1024
#define DBL_MIN_10_EXP  -307
#define DBL_MIN_EXP     -1021
#define DBL_MAX         0x1.fffffffffffffp1023
#define DBL_MIN         0x1.0p-1022
#define DBL_EPSILON     0x1.0p-52
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	**Macro for application**
DBL_DIG	CL_DBL_DIG
DBL_MANT_DIG	CL_DBL_MANT_DIG
DBL_MAX_10_EXP	CL_DBL_MAX_10_EXP
DBL_MAX_EXP	CL_DBL_MAX_EXP
DBL_MIN_10_EXP	CL_DBL_MIN_10_EXP

DBL_MIN_EXP	CL_DBL_MIN_EXP
DBL_MAX	CL_DBL_MAX
DBL_MIN	CL_DBL_MIN
DBL_EPSILSON	CL_DBL_EPSILON

The following constants are also available. They are of type double and are accurate within the precision of the double type.

Constant	Description
M_E	Value of *e*
M_LOG2E	Value of $\log_2 e$
M_LOG10E	Value of $\log_{10} e$
M_LN2	Value of $\log_e 2$
M_LN10	Value of $\log_e 10$
M_PI	Value of π
M_PI_2	Value of π / 2
M_PI_4	Value of π / 4
M_1_PI	Value of 1 / π
M_2_PI	Value of 2 / π
M_2_SQRTPI	Value of 2 / √π
M_SQRT2	Value of √2
M_SQRT1_2	Value of 1 / √2

6.13.3. Integer Functions

The following table describes the built-in integer functions that take scalar or vector arguments. The vector versions of the integer functions operate component-wise. The description is per-component.

We use the generic type name gentype to indicate that the function can take char, char{2|3|4|8|16}, uchar, uchar{2|3|4|8|16}, short, short{2|3|4|8|16}, ushort, ushort{2|3|4|8|16}, int, int{2|3|4|8|16}, uint, uint{2|3|4|8|16}, long, long{2|3|4|8|16} ulong, or ulong{2|3|4|8|16} as the type for the arguments. We use the generic type name ugentype to refer to unsigned versions of gentype. For example, if gentype is char4, ugentype is uchar4. We also use the generic type name sgentype to indicate that the function can take a scalar data type, i.e. char, uchar, short, ushort, int, uint, long, or ulong, as the type for the arguments. For built-in integer functions that take gentype and sgentype arguments, the gentype argument must be a vector or scalar version of the sgentype argument. For example, if sgentype is uchar, gentype must be uchar or uchar{2|3|4|8|16}. For vector versions, sgentype is implicitly widened to gentype as described for arithmetic operators.

For any specific use of a function, the actual type has to be the same for all arguments and the return type unless otherwise specified.

Table 10. Scalar and Vector Integer Argument Built-in Functions

Function	Description
ugentype **abs**(gentype *x*)	Returns |x|.
ugentype **abs_diff**(gentype *x*, gentype *y*)	Returns |x - y| without modulo overflow.
gentype **add_sat**(gentype *x*, gentype *y*)	Returns *x* + *y* and saturates the result.
gentype **hadd**(gentype *x*, gentype *y*)	Returns (*x* + *y*) >> 1. The intermediate sum does not modulo overflow.
gentype **rhadd**(gentype *x*, gentype *y*)[32]	Returns (*x* + *y* + 1) >> 1. The intermediate sum does not modulo overflow.
gentype **clamp**(gentype *x*, gentype *minval*, gentype *maxval*) gentype **clamp**(gentype *x*, sgentype *minval*, sgentype *maxval*)	Returns **min**(**max**(*x*, *minval*), *maxval*). Results are undefined if *minval* > *maxval*.
gentype **clz**(gentype *x*)	Returns the number of leading 0-bits in *x*, starting at the most significant bit position. If *x* is 0, returns the size in bits of the type of *x* or component type of *x*, if *x* is a vector.
gentype **ctz**(gentype *x*)	Returns the count of trailing 0-bits in *x*. If *x* is 0, returns the size in bits of the type of *x* or component type of *x*, if *x* is a vector.
gentype **mad_hi**(gentype *a*, gentype *b*, gentype *c*)	Returns **mul_hi**(*a*, *b*) + *c*.
gentype **mad_sat**(gentype *a*, gentype *b*, gentype *c*)	Returns *a* * *b* + *c* and saturates the result.
gentype **max**(gentype *x*, gentype *y*) gentype **max**(gentype *x*, sgentype *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*.
gentype **min**(gentype *x*, gentype *y*) gentype **min**(gentype *x*, sgentype *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*.
gentype **mul_hi**(gentype *x*, gentype *y*)	Computes *x* * *y* and returns the high half of the product of *x* and *y*.
gentype **rotate**(gentype *v*, gentype *i*)	For each element in *v*, the bits are shifted left by the number of bits given by the corresponding element in *i* (subject to the usual shift modulo rules). Bits shifted off the left side of the element are shifted back in from the right.
gentype **sub_sat**(gentype *x*, gentype *y*)	Returns *x* - *y* and saturates the result.
short **upsample**(char *hi*, uchar *lo*) ushort **upsample**(uchar *hi*, uchar *lo*) short*n* **upsample**(char*n* *hi*, uchar*n* *lo*) ushort*n* **upsample**(uchar*n* *hi*, uchar*n* *lo*)	*result*[i] = ((short)*hi*[i] << 8) | *lo*[i] *result*[i] = ((ushort)*hi*[i] << 8) | *lo*[i]
int **upsample**(short *hi*, ushort *lo*) uint **upsample**(ushort *hi*, ushort *lo*) int*n* **upsample**(short*n* *hi*, ushort*n* *lo*) uint*n* **upsample**(ushort*n* *hi*, ushort*n* *lo*)	*result*[i] = ((int)*hi*[i] << 16) | *lo*[i] *result*[i] = ((uint)*hi*[i] << 16) | *lo*[i]

long **upsample**(int *hi*, uint *lo*) ulong **upsample**(uint *hi*, uint *lo*) long*n* **upsample**(int*n* *hi*, uint*n* *lo*) ulong*n* **upsample**(uint*n* *hi*, uint*n* *lo*)	*result*[i] = ((long)*hi*[i] << 32) | *lo*[i] *result*[i] = ((ulong)*hi*[i] << 32) | *lo*[i]
gentype **popcount**(gentype *x*)	Returns the number of non-zero bits in *x*.

[32] Frequently vector operations need n + 1 bits temporarily to calculate a result. The **rhadd** instruction gives you an extra bit without needing to upsample and downsample. This can be a profound performance win.

The following table describes fast integer functions that can be used for optimizing performance of kernels. We use the generic type name `gentype` to indicate that the function can take `int`, `int2`, `int3`, `int4`, `int8`, `int16`, `uint`, `uint2`, `uint3`, `uint4`, `uint8` or `uint16` as the type for the arguments.

Table 11. Fast Integer Built-in Functions

Function	Description
gentype **mad24**(gentype *x*, gentype *y*, gentype z)	Multipy two 24-bit integer values *x* and *y* and add the 32-bit integer result to the 32-bit integer *z*. Refer to definition of **mul24** to see how the 24-bit integer multiplication is performed.
gentype **mul24**(gentype *x*, gentype *y*)	Multiply two 24-bit integer values *x* and *y*. *x* and *y* are 32-bit integers but only the low 24-bits are used to perform the multiplication. **mul24** should only be used when values in *x* and *y* are in the range [-2^{23}, $2^{23}-1$] if *x* and *y* are signed integers and in the range [0, $2^{24}-1$] if *x* and *y* are unsigned integers. If *x* and *y* are not in this range, the multiplication result is implementation-defined.

The macro names given in the following list must use the values specified. The values shall all be constant expressions suitable for use in `#if` preprocessing directives.

```
#define CHAR_BIT        8
#define CHAR_MAX        SCHAR_MAX
#define CHAR_MIN        SCHAR_MIN
#define INT_MAX         2147483647
#define INT_MIN         (-2147483647 - 1)
#define LONG_MAX        0x7fffffffffffffffL
#define LONG_MIN        (-0x7fffffffffffffffL - 1)
#define SCHAR_MAX       127
#define SCHAR_MIN       (-127 - 1)
#define SHRT_MAX        32767
#define SHRT_MIN        (-32767 - 1)
#define UCHAR_MAX       255
#define USHRT_MAX       65535
#define UINT_MAX        0xffffffff
#define ULONG_MAX       0xffffffffffffffffUL
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
`CHAR_BIT`	`CL_CHAR_BIT`
`CHAR_MAX`	`CL_CHAR_MAX`
`CHAR_MIN`	`CL_CHAR_MIN`
`INT_MAX`	`CL_INT_MAX`
`INT_MIN`	`CL_INT_MIN`
`LONG_MAX`	`CL_LONG_MAX`
`LONG_MIN`	`CL_LONG_MIN`
`SCHAR_MAX`	`CL_SCHAR_MAX`
`SCHAR_MIN`	`CL_SCHAR_MIN`
`SHRT_MAX`	`CL_SHRT_MAX`
`SHRT_MIN`	`CL_SHRT_MIN`
`UCHAR_MAX`	`CL_UCHAR_MAX`
`USHRT_MAX`	`CL_USHRT_MAX`
`UINT_MAX`	`CL_UINT_MAX`
`ULONG_MAX`	`CL_ULONG_MAX`

6.13.4. Common Functions[33]

[33] The **mix** and **smoothstep** functions can be implemented using contractions such as **mad** or **fma**.

The following table describes the list of built-in common functions. These all operate component-wise. The description is per-component. We use the generic type name `gentype` to indicate that the function can take `float`, `float2`, `float3`, `float4`, `float8`, `float16`, `double`, `double2`, `double3`, `double4`, `double8` or `double16` as the type for the arguments. We use the generic type name `gentypef` to indicate that the function can take `float`, `float2`, `float3`, `float4`, `float8`, or `float16` as the type for the arguments. We use the generic type name `gentyped` to indicate that the function can take `double`, `double2`, `double3`, `double4`, `double8` or `double16` as the type for the arguments.

The built-in common functions are implemented using the round to nearest even rounding mode.

Table 12. Scalar and Vector Argument Built-in Common Function Table

Function	Description
gentype **clamp**(gentype *x*, gentype *minval*, gentype *maxval*) gentypef **clamp**(gentypef *x*, float *minval*, float *maxval*) gentyped **clamp**(gentyped *x*, double *minval*, double *maxval*)	Returns **fmin**(**fmax**(*x*, *minval*), *maxval*). Results are undefined if *minval* > *maxval*.
gentype **degrees**(gentype *radians*)	Converts *radians* to degrees, i.e. (180 / π) * *radians*.

gentype **max**(gentype *x*, gentype *y*) gentypef **max**(gentypef *x*, float *y*) gentyped **max**(gentyped *x*, double *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*. If *x* or *y* are infinite or NaN, the return values are undefined.
gentype **min**(gentype *x*, gentype *y*) gentypef **min**(gentypef *x*, float *y*) gentyped **min**(gentyped *x*, double *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*. If *x* or *y* are infinite or NaN, the return values are undefined.
gentype **mix**(gentype *x*, gentype *y*, gentype *a*) gentypef **mix**(gentypef *x*, gentypef *y*, float *a*) gentyped **mix**(gentyped *x*, gentyped *y*, double *a*)	Returns the linear blend of *x* & *y* implemented as: *x* + (*y* - *x*) * *a* *a* must be a value in the range [0.0, 1.0]. If *a* is not in the range [0.0, 1.0], the return values are undefined.
gentype **radians**(gentype *degrees*)	Converts *degrees* to radians, i.e. (π / 180) * *degrees*.
gentype **step**(gentype *edge*, gentype *x*) gentypef **step**(float *edge*, gentypef *x*) gentyped **step**(double *edge*, gentyped *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0.

gentype **smoothstep**(gentype *edge0*, gentype *edge1*, gentype *x*)
gentypef **smoothstep**(float *edge0*, float *edge1*, gentypef *x*)
gentyped **smoothstep**(double *edge0*, double *edge1*, gentyped *x*)
	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1 when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition.
	
	This is equivalent to:
	
	`gentype t;`
	`t = clamp ((x - edge0) / (edge1 - edge0), 0, 1);`
	`return t * t * (3 - 2 * t);`
	
	Results are undefined if *edge0* >= *edge1* or if *x*, *edge0* or *edge1* is a NaN.

gentype **sign**(gentype *x*)	Returns 1.0 if *x* > 0, -0.0 if *x* = -0.0, +0.0 if *x* = +0.0, or -1.0 if *x* < 0. Returns 0.0 if *x* is a NaN.

6.13.5. Geometric Functions[34]

[34] The geometric functions can be implemented using contractions such as **mad** or **fma**.

The following table describes the list of built-in geometric functions. These all operate component-wise. The description is per-component. `floatn` is `float`, `float2`, `float3`, or `float4` and `doublen` is `double`, `double2`, `double3`, or `double4`. The built-in geometric functions are implemented using the round to nearest even rounding mode.

Table 13. Scalar and Vector Argument Built-in Geometric Function Table

Function	Description
float4 **cross**(float4 *p0*, float4 *p1*) float3 **cross**(float3 *p0*, float3 *p1*) double4 **cross**(double4 *p0*, double4 *p1*) double3 **cross**(double3 *p0*, double3 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* component of `float4` result returned will be 0.0.
float **dot**(float*n* *p0*, float*n* *p1*) double **dot**(double*n* *p0*, double*n* *p1*)	Compute dot product.
float **distance**(float*n* *p0*, float*n* *p1*) double **distance**(double*n* *p0*, double*n* *p1*)	Returns the distance between *p0* and *p1*. This is calculated as **length**(*p0* - *p1*).
float **length**(float*n* *p*) double **length**(double*n* *p*)	Return the length of vector *p*, i.e., $\sqrt{p.x^2 + p.y^2 + \ldots}$
float*n* **normalize**(float*n* *p*) double*n* **normalize**(double*n* *p*)	Returns a vector in the same direction as *p* but with a length of 1.
float **fast_distance**(float*n* *p0*, float*n* *p1*)	Returns **fast_length**(*p0* - *p1*).
float **fast_length**(float*n* *p*)	Returns the length of vector *p* computed as: **half_sqrt**($p.x^2 + p.y^2 + \ldots$)

float*n* **fast_normalize**(float*n* *p*)	Returns a vector in the same direction as *p* but with a length of 1. **fast_normalize** is computed as:
	
	p * **half_rsqrt**($p.x^2 + p.y^2 + \ldots$)
	
	The result shall be within 8192 ulps error from the infinitely precise result of
	
	`if (all(p == 0.0f))`
	`    result = p;`
	`else`
	`    result = p / sqrt(p.x^2 + p.y^2 + ...);`
	
	with the following exceptions:
	
	1. If the sum of squares is greater than `FLT_MAX` then the value of the floating-point values in the result vector are undefined.
	2. If the sum of squares is less than `FLT_MIN` then the implementation may return back *p*.
	3. If the device is in "denorms are flushed to zero" mode, individual operand elements with magnitude less than **sqrt**(`FLT_MIN`) may be flushed to zero before proceeding with the calculation.

6.13.6. Relational Functions

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with scalar and vector built-in types and produce a scalar or vector signed integer result respectively.

The functions[35] described in the following table can be used with built-in scalar or vector types as arguments and return a scalar or vector integer result. The argument type `gentype` refers to the following built-in types: `char`, `charn`, `uchar`, `ucharn`, `short`, `shortn`, `ushort`, `ushortn`, `int`, `intn`, `uint`, `uintn`, `long`, `longn`, `ulong`, `ulongn`, `float`, `floatn`, `double`, and `doublen`. The argument type `igentype` refers to the built-in signed integer types i.e. `char`, `charn`, `short`, `shortn`, `int`, `intn`, `long` and `longn`. The argument type `ugentype` refers to the built-in unsigned integer types i.e. **uchar**, **uchar*n***, **ushort**, **ushort*n***, `uint`, `uintn`, `ulong` and `ulongn`. *n* is 2, 3, 4, 8, or 16.

[35] If an implementation extends this specification to support IEEE-754 flags or exceptions, then all builtin functions defined in the following table shall proceed without raising the *invalid* floating-point exception when one or more of the operands are NaNs.

The functions **isequal**, **isnotequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isordered**, **isunordered** and **signbit** described in the following table shall return a 0 if the specified relation is *false* and a 1 if the specified relation is *true* for scalar argument types. These functions shall return a 0 if the specified relation is *false* and a -1 (i.e. all bits set) if the specified relation is *true* for vector argument types.

The relational functions **isequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, and **islessgreater** always return 0 if either argument is not a number (NaN). **isnotequal** returns 1 if one or both arguments are not a number (NaN) and the argument type is a scalar and returns -1 if one or both arguments are not a number (NaN) and the argument type is a vector.

Table 14. Scalar and Vector Relational Functions

Function	Description
int **isequal**(float *x*, float *y*) int*n* **isequal**(float*n* *x*, float*n* *y*) int **isequal**(double *x*, double *y*) long*n* **isequal**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of *x* == *y*.
int **isnotequal**(float *x*, float *y*) int*n* **isnotequal**(float*n* *x*, float*n* *y*) int **isnotequal**(double *x*, double *y*) long*n* **isnotequal**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of *x* != *y*.
int **isgreater**(float *x*, float *y*) int*n* **isgreater**(float*n* *x*, float*n* *y*) int **isgreater**(double *x*, double *y*) long*n* **isgreater**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of *x* > *y*.
int **isgreaterequal**(float *x*, float *y*) int*n* **isgreaterequal**(float*n* *x*, float*n* *y*) int **isgreaterequal**(double *x*, double *y*) long*n* **isgreaterequal**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of *x* >= *y*.
int **isless**(float *x*, float *y*) int*n* **isless**(float*n* *x*, float*n* *y*) int **isless**(double *x*, double *y*) long*n* **isless**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of *x* < *y*.
int **islessequal**(float *x*, float *y*) int*n* **islessequal**(float*n* *x*, float*n* *y*) int **islessequal**(double *x*, double *y*) long*n* **islessequal**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of *x* <= *y*.
int **islessgreater**(float *x*, float *y*) int*n* **islessgreater**(float*n* *x*, float*n* *y*) int **islessgreater**(double *x*, double *y*) long*n* **islessgreater**(double*n* *x*, double*n* *y*)	Returns the component-wise compare of (*x* < *y*) || (*x* > *y*) .
int **isfinite**(float) int*n* **isfinite**(float*n*) int **isfinite**(double) long*n* **isfinite**(double*n*)	Test for finite value.
int **isinf**(float) int*n* **isinf**(float*n*) int **isinf**(double) long*n* **isinf**(double*n*)	Test for infinity value (positive or negative).

int **isnan**(float) int*n* **isnan**(float*n*) int **isnan**(double) long*n* **isnan**(double*n*)	Test for a NaN.
int **isnormal**(float) int*n* **isnormal**(float*n*) int **isnormal**(double) long*n* **isnormal**(double*n*)	Test for a normal value.
int **isordered**(float *x*, float *y*) int*n* **isordered**(float*n* *x*, float*n* *y*) int **isordered**(double *x*, double *y*) long*n* **isordered**(double*n* *x*, double*n* *y*)	Test if arguments are ordered. **isordered**() takes arguments *x* and *y*, and returns the result **isequal**(*x*, *x*) && **isequal**(*y*, *y*).
int **isunordered**(float *x*, float *y*) int*n* **isunordered**(float*n* *x*, float*n* *y*) int **isunordered**(double *x*, double *y*) long*n* **isunordered**(double*n* *x*, double*n* *y*)	Test if arguments are unordered. **isunordered**() takes arguments *x* and *y*, returning non-zero if *x* or *y* is NaN, and zero otherwise.
int **signbit**(float) int*n* **signbit**(float*n*) int **signbit**(double) long*n* **signbit**(double*n*)	Test for sign bit. The scalar version of the function returns a 1 if the sign bit in the float is set else returns 0. The vector version of the function returns the following for each component in `floatn`: -1 (i.e all bits set) if the sign bit in the float is set else returns 0.
int **any**(igentype *x*)	Returns 1 if the most significant bit in any component of *x* is set; otherwise returns 0.
int **all**(igentype *x*)	Returns 1 if the most significant bit in all components of *x* is set; otherwise returns 0.
gentype **bitselect**(gentype *a*, gentype *b*, gentype *c*)	Each bit of the result is the corresponding bit of *a* if the corresponding bit of *c* is 0. Otherwise it is the corresponding bit of *b*.
gentype **select**(gentype *a*, gentype *b*, igentype *c*) gentype **select**(gentype *a*, gentype *b*, ugentype *c*)	For each component of a vector type, *result[i]* = if MSB of *c[i]* is set ? *b[i]* : *a[i]*. For a scalar type, *result* = *c* ? *b* : *a*. `igentype` and `ugentype` must have the same number of elements and bits as `gentype`[36].

[36] The above definition means that the behavior of select and the ternary operator for vector and scalar types is dependent on different interpretations of the bit pattern of *c*.

6.13.7. Vector Data Load and Store Functions

The following table describes the list of supported functions that allow you to read and write vector types from a pointer to memory. We use the generic type `gentype` to indicate the built-in data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float` or `double`. We use the generic type name `gentypen` to represent n-element vectors of `gentype` elements. We use the type name `halfn` to represent n-element vectors of half elements[37]. The suffix *n* is also used in the function names (i.e. **vload*n***, **vstore*n*** etc.), where *n* = 2, 3, 4, 8 or 16.

[37] The `halfn` type is only defined by the **cl_khr_fp16** extension described in section 9.4 of the OpenCL 2.0 Extension Specification.

Table 15. Vector Data Load and Store Functions[38]

Function	Description
gentype*n* **vload*n***(size_t *offset*, const gentype **p*) gentype*n* **vload*n***(size_t *offset*, const constant gentype **p*)	Return `sizeof(gentypen)` bytes of data, where the first `(n * sizeof(gentype))` bytes are read from the address computed as `(p + (offset * n))`. The computed address must be 8-bit aligned if `gentype` is `char` or `uchar`; 16-bit aligned if `gentype` is `short` or `ushort`; 32-bit aligned if `gentype` is `int`, `uint`, or `float`; and 64-bit aligned if `gentype` is `long` or `ulong`.
void **vstore*n***(gentype*n* *data*, size_t *offset*, gentype **p*)	Write `n * sizeof(gentype)` bytes given by *data* to the address computed as `(p + (offset * n))`. The computed address must be 8-bit aligned if `gentype` is `char` or `uchar`; 16-bit aligned if `gentype` is `short` or `ushort`; 32-bit aligned if `gentype` is `int`, `uint`, or `float`; and 64-bit aligned if `gentype` is `long` or `ulong`.
float **vload_half**(size_t *offset*, const half **p*) float **vload_half**(size_t *offset*, const constant half **p*)	Read `sizeof(half)` bytes of data from the address computed as `(p + offset)`. The data read is interpreted as a `half` value. The `half` value is converted to a `float` value and the `float` value is returned. The computed read address must be 16-bit aligned.
float*n* **vload_half*n***(size_t *offset*, const half **p*) float*n* **vload_half*n***(size_t *offset*, const constant half **p*)	Read `(n * sizeof(half))` bytes of data from the address computed as `(p + (offset * n))`. The data read is interpreted as a `halfn` value. The `halfn` value read is converted to a `floatn` value and the `floatn` value is returned. The computed read address must be 16-bit aligned.
void **vstore_half**(float *data*, size_t *offset*, half **p*) void **vstore_half*rte*(float _data**, size_t *offset*, half **p*) void **vstore_half*rtz*(float _data**, size_t *offset*, half **p*) void **vstore_half*rtp*(float _data**, size_t *offset*, half **p*) void **vstore_half*rtn*(float _data**, size_t *offset*, half **p*)	The `float` value given by *data* is first converted to a `half` value using the appropriate rounding mode. The `half` value is then written to the address computed as `(p + offset)`. The computed address must be 16-bit aligned. **vstore_half** uses the default rounding mode. The default rounding mode is round to nearest even.

void **vstore_half*n***(float*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rte***(float*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rtz***(float*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rtp***(float*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rtn***(float*n* *data*, size_t *offset*, half *_p_)	The `floatn` value given by *data* is converted to a `halfn` value using the appropriate rounding mode. `n * sizeof(half)` bytes from the `halfn` value are then written to the address computed as `(p + (offset * n))`. The computed address must be 16-bit aligned. **vstore_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.
void **vstore_half**(double *data*, size_t *offset*, half *_p_) void **vstore_half*rte*(double _data**, size_t *offset*, half *_p_) void **vstore_half*rtz*(double _data**, size_t *offset*, half *_p_) void **vstore_half*rtp*(double _data**, size_t *offset*, half *_p_) void **vstore_half*rtn*(double _data**, size_t *offset*, half *_p_)	The `double` value given by *data* is first converted to a `half` value using the appropriate rounding mode. The `half` value is then written to the address computed as `(p + offset)`. The computed address must be 16-bit aligned. **vstore_half** uses the default rounding mode. The default rounding mode is round to nearest even.
void **vstore_half*n***(double*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rte***(double*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rtz***(double*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rtp***(double*n* *data*, size_t *offset*, half *_p_) void **vstore_half*n*_*rtn***(double*n* *data*, size_t *offset*, half *_p_)	The `doublen` value given by *data* is converted to a `halfn` value using the appropriate rounding mode. `n * sizeof(half)` bytes from the `halfn` value are then written to the address computed as `(p + (offset * n))`. The computed address must be 16-bit aligned. **vstore_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.
float*n* **vloada_half*n***(size_t *offset*, const half *_p_) float*n* **vloada_half*n***(size_t *offset*, const constant half *_p_)	For n = 2, 4, 8 and 16, read `sizeof(halfn)` bytes of data from the address computed as (*p* + (*offset* * *n*)). The data read is interpreted as a `halfn` value. The `halfn` value read is converted to a `floatn` value and the `floatn` value is returned. The computed address must be aligned to `sizeof(halfn)` bytes. For n = 3, **vloada_half3** reads a `half3` from the address computed as `(p + (offset * 4))` and returns a `float3`. The computed address must be aligned to `sizeof(half)` * 4 bytes.

void **vstorea_half*n***(float*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rte***(float*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rtz***(float*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rtp***(float*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rtn***(float*n* *data*, size_t *offset*, half **p*)	The `floatn` value given by *data* is converted to a `halfn` value using the appropriate rounding mode. For n = 2, 4, 8 and 16, the `halfn` value is written to the address computed as `(p + (offset * n))`. The computed address must be aligned to `sizeof(halfn)` bytes. For n = 3, the `half3` value is written to the address computed as `(p + (offset * 4))`. The computed address must be aligned to `sizeof(half) * 4` bytes. **vstorea_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.
void **vstorea_half*n***(double*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rte***(double*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rtz***(double*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rtp***(double*n* *data*, size_t *offset*, half **p*) void **vstorea_half*n*_*rtn***(double*n* *data*, size_t *offset*, half **p*)	The `doublen` value is converted to a `halfn` value using the appropriate rounding mode. For n = 2, 4, 8 or 16, the `halfn` value is written to the address computed as `(p + (offset * n))`. The computed address must be aligned to `sizeof(halfn)` bytes. For n = 3, the `half3` value is written to the address computed as `(p + (offset * 4))`. The computed address must be aligned to `sizeof(half) * 4` bytes. **vstorea_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.

[38] **vload3** and **vload_half3** read (*x*,*y*,*z*) components from address `(p + (offset * 3))` into a 3-component vector. **vstore3** and **vstore_half3** write (*x*,*y*,*z*) components from a 3-component vector to address `(p + (offset * 3))`. In addition, **vloada_half3** reads (*x*,*y*,*z*) components from address `(p + (offset * 4))` into a 3-component vector and **vstorea_half3** writes (*x*,*y*,*z*) components from a 3-component vector to address `(p + (offset * 4))`. Whether **vloada_half3** and **vstorea_half3** read/write padding data between the third vector element and the next alignment boundary is implementation defined. **vloada_** and **vstorea_** variants are provided to access data that is aligned to the size of the vector, and are intended to enable performance on hardware that can take advantage of the increased alignment.

The results of vector data load and store functions are undefined if the address being read from or written to is not correctly aligned as described in Vector Data Load and Store Functions[38]. The pointer argument p can be a pointer to `global`, `local`, or `private` memory for store functions described in Vector Data Load and Store Functions[38]. The pointer argument p can be a pointer to `global`, `local`, `constant`, or `private` memory for load functions described in Vector Data Load and Store Functions[38].

The vector data load and store functions variants that take pointer arguments which point to the generic address space are also supported.

6.13.8. Synchronization Functions

The OpenCL C programming language implements the following synchronization functions.

Table 16. Built-in Synchronization Functions

Function	Description
void **work_group_barrier**[39](cl_mem_fence_flags *flags*) void **work_group_barrier**(cl_mem_fence_flags *flags*, memory_scope *scope*[40])	All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the **work_group_barrier**. This function must be encountered by all work-items in a work-group executing the kernel. These rules apply to ND-ranges implemented with uniform and non-uniform work-groups. If **work_group_barrier** is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the **work_group_barrier**. If **work_group_barrier** is inside a loop, all work-items must execute the **work_group_barrier** for each iteration of the loop before any are allowed to continue execution beyond the **work_group_barrier**. The **work_group_barrier** function also supports a variant that specifies the memory scope. For the **work_group_barrier** variant that does not take a memory scope, the *scope* is `memory_scope_work_group`. The *scope* argument specifies whether the memory accesses of work-items in the work-group to memory address space(s) identified by *flags* become visible to all work-items in the work-group, the device or all SVM devices. The **work_group_barrier** function can also be used to specify which memory operations, i.e. to `global` memory, `local` memory or images become visible to the appropriate memory scope identified by *scope*. The *flags* argument specifies the memory address spaces. This is a bitfield and can be set to 0 or a combination of the following values ORed together. When these flags are OR'ed together the **work_group_barrier** acts as a combined barrier for all address spaces specified by the flags ordering memory accesses both within and across the specified address spaces. `CLK_LOCAL_MEM_FENCE` - The **work_group_barrier** function will ensure that all local memory accesses become visible to all work-items in the work-group. Note that the value of *scope* is ignored as the memory scope is always `memory_scope_work_group`.

[39] The built-in function **barrier** has been renamed **work_group_barrier**. For backward compatibility, **barrier** is also supported.

[40] Refer to the description of `memory_scope`.

6.13.9. Address Space Qualifier Functions

The OpenCL C programming language implements the following address space qualifier functions. We use the generic type name `gentype` to indicate any of the built-in data types supported by OpenCL C or a user defined type.

Table 17. Built-in Address Space Qualifier Functions

Function	Description
global gentype * **to_global**(gentype **ptr*) const global gentype * **to_global**(const gentype **ptr*)	Returns a pointer that points to a region in the `global` address space if **to_global** can cast *ptr* to the `global` address space. Otherwise it returns `NULL`.
local gentype * **to_local**(gentype **ptr*) const local gentype * **to_local**(const gentype * *ptr*)	Returns a pointer that points to a region in the `local` address space if **to_local** can cast *ptr* to the local address space. Otherwise it returns `NULL`.
private gentype * **to_private**(gentype **ptr*) const private gentype * **to_private**(const gentype **ptr*)	Returns a pointer that points to a region in the `private` address space if **to_private** can cast *ptr* to the `private` address space. Otherwise it returns `NULL`.
cl_mem_fence_flags **get_fence**(gentype **ptr*) cl_mem_fence_flags **get_fence**(const gentype **ptr*)	Returns a valid memory fence value for *ptr*.

6.13.10. Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch

The OpenCL C programming language implements the following functions that provide asynchronous copies between `global` and local memory and a prefetch from `global` memory.

We use the generic type name `gentype` to indicate the built-in data types char, `char{2|3`[41]`|4|8|16}`, `uchar`, `uchar{2|3|4|8|16}`, `short`, `short{2|3|4|8|16}`, `ushort`, `ushort{2|3|4|8|16}`, `int`, `int{2|3|4|8|16}`, `uint`, `uint{2|3|4|8|16}`, `long`, `long{2|3|4|8|16}`, `ulong`, `ulong{2|3|4|8|16}`, `float`, `float{2|3|4|8|16}`, or `double`, `double{2|3|4|8|16}` as the type for the arguments unless otherwise stated.

[41] **async_work_group_copy** and **async_work_group_strided_copy** for 3-component vector types behave as **async_work_group_copy** and **async_work_group_strided_copy** respectively for 4-component vector types.

Table 18. Built-in Async Copy and Prefetch Functions

Function	Description

event_t **async_work_group_copy**(*local gentype *dst, const* global gentype **src*, size_t *num_gentypes*, event_t *event*) event_t **async_work_group_copy**(*global gentype *dst, const* local gentype **src*, size_t *num_gentypes*, event_t *event*)	Perform an async copy of *num_gentypes* gentype elements from *src* to *dst*. The async copy is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. This rule applies to ND-ranges implemented with uniform and non-uniform work-groups. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. 0 can be implicitly and explicitly cast to `event_t` type. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy.

event_t **async_work_group_strided_copy**(*local gentype *dst, const* global gentype **src*, size_t *num_gentypes*, size_t *src_stride*, event_t *event*) event_t **async_work_group_strided_copy**(*global gentype *dst, const* local gentype **src*, size_t *num_gentypes*, size_t *dst_stride*, event_t *event*)	Perform an async gather of *num_gentypes* `gentype` elements from *src* to *dst*. The *src_stride* is the stride in elements for each `gentype` element read from *src*. The *dst_stride* is the stride in elements for each `gentype` element written to *dst*. The async gather is performed by all work-items in a work-group. This built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. This rule applies to ND-ranges implemented with uniform and non-uniform work-groups Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_strided_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. 0 can be implicitly and explicitly cast to event_t type. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy. The behavior of **async_work_group_strided_copy** is undefined if *src_stride* or *dst_stride* is 0, or if the *src_stride* or *dst_stride* values cause the *src* or *dst* pointers to exceed the upper bounds of the address space during the copy.
void **wait_group_events**(int *num_events*, event_t **event_list*)	Wait for events that identify the **async_work_group_copy** operations to complete. The event objects specified in *event_list* will be released after the wait is performed. This function must be encountered by all work-items in a work-group executing the kernel with the same *num_events* and event objects specified in *event_list*; otherwise the results are undefined. This rule applies to ND-ranges implemented with uniform and non-uniform work-groups

void **prefetch**(const *_global gentype *_p*, size_t *num_gentypes*)	Prefetch `num_gentypes * sizeof(gentype)` bytes into the global cache. The prefetch instruction is applied to a work-item in a work-group and does not affect the functional behavior of the kernel.

The kernel must wait for the completion of all async copies using the **wait_group_events** built-in function before exiting; otherwise the behavior is undefined.

6.13.11. Atomic Functions

The OpenCL C programming language implements a subset of the C11 atomics (refer to section 7.17 of the C11 Specification) and synchronization operations. These operations play a special role in making assignments in one work-item visible to another. A synchronization operation on one or more memory locations is either an acquire operation, a release operation, or both an acquire and release operation[42]. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations which have special characteristics.

[42] The C11 consume operation is not supported.

The types include

```
memory_order
```

which is an enumerated type whose enumerators identify memory ordering constraints;

```
memory_scope
```

which is an enumerated type whose enumerators identify scope of memory ordering constraints;

```
atomic_flag
```

which is a 32-bit integer type representing a lock-free, primitive atomic flag; and several atomic analogs of integer types.

In the following operation definitions:

- An A refers to one of the atomic types.
- A C refers to its corresponding non-atomic type.
- An M refers to the type of the other argument for arithmetic operations. For atomic integer types, M is C.
- The functions not ending in explicit have the same semantics as the corresponding explicit function with `memory_order_seq_cst` for the `memory_order` argument.
- The functions that do not have `memory_scope` argument have the same semantics as the corresponding functions with the `memory_scope` argument set to `memory_scope_device`.

> With fine-grained system SVM, sharing happens at the granularity of individual loads and stores anywhere in host memory. Memory consistency is always guaranteed at synchronization points, but to obtain finer control over consistency, the OpenCL atomics functions may be used to ensure that the updates to individual data values made by one unit of execution are visible to other execution units. In particular, when a host thread needs fine control over the consistency of memory that is shared with one or more OpenCL devices, it must use atomic and fence operations that are compatible with the C11 atomic operations.
>
> We can't require C11 atomics since host programs can be implemented in other programming languages and versions of C or C++, but we do require that the host programs use atomics and that those atomics be compatible with those in C11.

The `ATOMIC_VAR_INIT` macro

The `ATOMIC_VAR_INIT` macro expands to a token sequence suitable for initializing an atomic object of a type that is initialization-compatible with value. An atomic object with automatic storage duration that is not explicitly initialized using `ATOMIC_VAR_INIT` is initially in an indeterminate state; however, the default (zero) initialization for objects with `static` storage duration is guaranteed to produce a valid state.

```
#define ATOMIC_VAR_INIT(C value)
```

This macro can only be used to initialize atomic objects that are declared in program scope in the `global` address space.

Examples:

```
global atomic_int guide = ATOMIC_VAR_INIT(42);
```

Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data-race.

The atomic_init function

The atomic_init function non-atomically initializes the atomic object pointed to by obj to the value value.

```
void atomic_init(volatile A *obj, C value)
```

Examples:

```
local atomic_int local_guide;
if (get_local_id(0) == 0)
    atomic_init(&guide, 42);
work_group_barrier(CLK_LOCAL_MEM_FENCE);
```

Order and Consistency

The enumerated type `memory_order` specifies the detailed regular (non-atomic) memory synchronization operations as defined in section 5.1.2.4 of the C11 Specification, and may provide for operation ordering. Its enumeration constants are as follows:

`memory_order_relaxed`

`memory_order_acquire`

`memory_order_release`

`memory_order_acq_rel`

`memory_order_seq_cst`

The `memory_order` can be used when performing atomic operations to `global` or `local` memory.

Memory Scope

The enumerated type `memory_scope` specifies whether the memory ordering constraints given by `memory_order` apply to work-items in a work-group or work-items of a kernel(s) executing on the device or across devices (in the case of shared virtual memory). Its enumeration constants are as follows:

`memory_scope_work_item`[44]

`memory_scope_work_group`

`memory_scope_device`

`memory_scope_all_svm_devices`

[44] This value for `memory_scope` can only be used with `atomic_work_item_fence` with flags set to `CLK_IMAGE_MEM_FENCE`.

The memory scope should only be used when performing atomic operations to global memory. Atomic operations to `local` memory only guarantee memory ordering in the work-group not across work-groups and therefore ignore the `memory_scope` value.

Fences

The following fence operations are supported.

```
void atomic_work_item_fence(cl_mem_fence_flags flags,
                            memory_order order,
                            memory_scope scope)
```

`flags` must be set to `CLK_GLOBAL_MEM_FENCE`, `CLK_LOCAL_MEM_FENCE`, `CLK_IMAGE_MEM_FENCE` or a combination of these values ORed together; otherwise the behavior is undefined. The behavior of calling `atomic_work_item_fence` with `CLK_IMAGE_MEM_FENCE` ORed together with either `CLK_GLOBAL_MEM_FENCE` or `CLK_LOCAL_MEM_FENCE` is equivalent to calling `atomic_work_item_fence` individually for `CLK_IMAGE_MEM_FENCE` and the other flags. Passing both `CLK_GLOBAL_MEM_FENCE` and `CLK_LOCAL_MEM_FENCE` to `atomic_work_item_fence` will synchronize memory operations to both `local` and `global` memory through some shared atomic action, as described in section 3.3.6.2 of the OpenCL Specification.

Depending on the value of order, this operation:

- has no effects, if *order* == `memory_order_relaxed`.
- is an acquire fence, if *order* == `memory_order_acquire`.
- is a release fence, if *order* == `memory_order_release`.
- is both an acquire fence and a release fence, if *order* == `memory_order_acq_rel`.
- is a sequentially consistent acquire and release fence, if *order* == `memory_order_seq_cst`.

For images declared with the `read_write` qualifier, the `atomic_work_item_fence` must be called to make sure that writes to the image by a work-item become visible to that work-item on subsequent reads to that image by that work-item.

Atomic integer and floating-point types

The list of supported atomic type names are:

`atomic_int`

`atomic_uint`

`atomic_long`[45]

`atomic_ulong`

`atomic_float`

`atomic_double`[46]

`atomic_intptr_t`[47]

`atomic_uintptr_t`

`atomic_size_t`

`atomic_ptrdiff_t`

[45] The atomic_long and atomic_ulong types are supported if the **cl_khr_int64_base_atomics** and **cl_khr_int64_extended_atomics** extensions are supported and have been enabled.

[46] The `atomic_double` type is only supported if double precision is supported and the **cl_khr_int64_base_atomics** and **cl_khr_int64_extended_atomics** extensions are supported and have been enabled.

[47] If the device address space is 64-bits, the data types `atomic_intptr_t`, `atomic_uintptr_t`, `atomic_size_t` and `atomic_ptrdiff_t` are supported if the **cl_khr_int64_base_atomics** and **cl_khr_int64_extended_atomics** extensions are supported and have been enabled.

Arguments to a kernel can be declared to be a pointer to the above atomic types or the atomic_flag type.

The representation of atomic integer, floating-point and pointer types have the same size as their corresponding regular types. The atomic_flag type must be implemented as a 32-bit integer.

Operations on atomic types

There are only a few kinds of operations on atomic types, though there are many instances of those kinds. This section specifies each general kind.

The atomic_store functions

```
void atomic_store(volatile A *object, C desired)

void atomic_store_explicit(volatile A *object,
                           C desired,
                           memory_order order)

void atomic_store_explicit(volatile A *object,
                           C desired,
                           memory_order order,
                           memory_scope scope)
```

The *order* argument shall not be `memory_order_acquire`, nor `memory_order_acq_rel`. Atomically replace the value pointed to by *object* with the value of *desired*. Memory is affected according to the value of *order*.

The atomic_load functions

```
C atomic_load(volatile A *object)

C atomic_load_explicit(volatile A *object,
                       memory_order order)

C atomic_load_explicit(volatile A *object,
                       memory_order order,
                       memory_scope scope)
```

The *order* argument shall not be `memory_order_release` nor `memory_order_acq_rel`. Memory is affected according to the value of *order*. Atomically returns the value pointed to by *object*.

The atomic_exchange functions

```
C atomic_exchange(volatile A *object, C desired)

C atomic_exchange_explicit(volatile A *object,
                           C desired,
                           memory_order order)

C atomic_exchange_explicit(volatile A *object,
                           C desired,
                           memory_order order,
                           memory_scope scope)
```

Atomically replace the value pointed to by object with desired. Memory is affected according to the value of order. These operations are read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). Atomically returns the value pointed to by object immediately before the effects.

The atomic_compare_exchange functions

```
bool atomic_compare_exchange_strong(
    volatile A *object,
    C *expected, C desired)

bool atomic_compare_exchange_strong_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure)

bool atomic_compare_exchange_strong_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)

bool atomic_compare_exchange_weak(
    volatile A *object,
    C *expected, C desired)

bool atomic_compare_exchange_weak_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure)

bool atomic_compare_exchange_weak_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
```

The *failure* argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall be no stronger than the `success` argument. Atomically, compares the value pointed to by object for equality with that in expected, and if *true*, replaces the value pointed to by `object` with `desired`, and if *false*, updates the value in `expected` with the value pointed to by `object`. Further, if the comparison is *true*, memory is affected according to the value of `success`, and if the comparison is *false*, memory is affected according to the value of `failure`. If the comparison is *true*, these operations are atomic read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). Otherwise, these operations are atomic load operations.

The effect of the compare-and-exchange operations is

```
if (memcmp(object, expected, sizeof(*object) == 0)
    memcpy(object, &desired, sizeof(*object));
else
    memcpy(expected, object, sizeof(*object));
```

The weak compare-and-exchange operations may fail spuriously[48]. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return zero and store back to `expected` the same memory contents that were originally there.

[48] This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines.

These generic functions return the result of the comparison.

The `atomic_fetch` and modify functions

The following operations perform arithmetic and bitwise computations. All of these operations are applicable to an object of any atomic integer type. The key, operator, and computation correspondence is given in table below:

key	op	computation
`add`	+	addition
`sub`	-	subtraction
`or`	|	bitwise inclusive or
`xor`	^	bitwise exclusive or
`and`	&	bitwise and
`min`	**min**	compute min
`max`	**max**	compute max

For **atomic_fetch** and modify functions with **key** = `add` or `sub` on atomic types `atomic_intptr_t` and `atomic_uintptr_t`, `M` is `ptrdiff_t`. For **atomic_fetch** and modify functions with **key** = `or`, `xor`, `and`, `min` and `max` on atomic types `atomic_intptr_t` and `atomic_uintptr_t`, `M` is `intptr_t` and `uintptr_t`.

```
C atomic_fetch_key(volatile A *object, M operand)

C atomic_fetch_key_explicit(volatile A *object,
                            M operand,
                            memory_order order)

C atomic_fetch_key_explicit(volatile A *object,
                            M operand,
                            memory_order order,
                            memory_scope scope)
```

Atomically replaces the value pointed to by object with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior. Returns atomically the value pointed to by `object` immediately before the effects.

Atomic flag type and operations

The atomic_flag type provides the classic test-and-set functionality. It has two states, *set* (value is non-zero) and *clear* (value is 0). Operations on an object of type atomic_flag shall be lock free.

The macro `ATOMIC_FLAG_INIT` may be used to initialize an atomic_flag to the *clear* state. An atomic_flag that is not explicitly initialized with `ATOMIC_FLAG_INIT` is initially in an indeterminate state.

This macro can only be used for atomic objects that are declared in program scope in the `global` address space with the `atomic_flag` type.

Example:

```
global atomic_flag guard = `ATOMIC_FLAG_INIT`;
```

The atomic_flag_test_and_set functions

```
bool atomic_flag_test_and_set(
    volatile atomic_flag *object)

bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag *object,
    memory_order order)

bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag *object,
    memory_order order,
    memory_scope scope)
```

Atomically sets the value pointed to by `object` to *true*. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). Returns atomically the value of the `object` immediately before the effects.

The atomic_flag_clear functions

```
void atomic_flag_clear(volatile atomic_flag *object)

void atomic_flag_clear_explicit(
    volatile atomic_flag *object,
    memory_order order)

void atomic_flag_clear_explicit(
    volatile atomic_flag *object,
    memory_order order,
    memory_scope scope)
```

The `order` argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`. Atomically sets the value pointed to by object to false. Memory is affected according to the value of order.

Restrictions

- All operations on atomic types must be performed using the built-in atomic functions. C11 and C++11 support operators on atomic types. OpenCL C does not support operators with atomic types. Using atomic types with operators should result in a compilation error.
- The `atomic_bool`, `atomic_char`, `atomic_uchar`, `atomic_short`, `atomic_ushort`, `atomic_intmax_t` and `atomic_uintmax_t` types are not supported by OpenCL C.
- OpenCL C requires that the built-in atomic functions on atomic types are lock-free.
- The _Atomic type specifier and _Atomic type qualifier are not supported by OpenCL C.
- The behavior of atomic operations where pointer arguments to the atomic functions refers to an atomic type in the `private` address space is undefined.

6.13.12. Miscellaneous Vector Functions

The OpenCL C programming language implements the following additional built-in vector functions. We use the generic type name `gentypen` (or `gentypem`) to indicate the built-in data types `char{2|4|8|16}`, `uchar{2|4|8|16}`, `short{2|4|8|16}`, `ushort{2|4|8|16}`, `half{2|4|8|16}`[49], `int{2|4|8|16}`, `uint{2|4|8|16}`, `long{2|4|8|16}`, `ulong{2|4|8|16}`, `float{2|4|8|16}`, or `double{2|4|8|16}`[50] as the type for the arguments unless otherwise stated. We use the generic name `ugentypen` to indicate the built-in unsigned integer data types.

[49] Only if the **cl_khr_fp16** extension is supported and has been enabled.

[50] Only if double precision is supported.

Table 19. Built-in Miscellaneous Vector Functions

Function	Description
int **vec_step**(gentype*n a*) int **vec_step**(char3 *a*) int **vec_step**(uchar3 *a*) int **vec_step**(short3 *a*) int **vec_step**(ushort3 *a*) int **vec_step**(half3 *a*) int **vec_step**(int3 *a*) int **vec_step**(uint3 *a*) int **vec_step**(long3 *a*) int **vec_step**(ulong3 *a*) int **vec_step**(float3 *a*) int **vec_step**(double3 *a*) int **vec_step**(*type*)	The **vec_step** built-in function takes a built-in scalar or vector data type argument and returns an integer value representing the number of elements in the scalar or vector. For all scalar types, **vec_step** returns 1. The **vec_step** built-in functions that take a 3-component vector return 4. **vec_step** may also take a pure type as an argument, e.g. **vec_step**(float2)

gentype*n* **shuffle**(gentype*m* *x*, ugentype*n* *mask*)
gentype*n* **shuffle2**(gentype*m* *x*, gentype*m* *y*, ugentype*n* *mask*)
	The **shuffle** and **shuffle2** built-in functions construct a permutation of elements from one or two input vectors respectively that are of the same type, returning a vector with the same element type as the input and length that is the same as the shuffle mask. The size of each element in the *mask* must match the size of each element in the result. For **shuffle**, only the **ilogb**(2_m_-1) least significant bits of each *mask* element are considered. For **shuffle2**, only the **ilogb**(2_m_-1)+1 least significant bits of each *mask* element are considered. Other bits in the mask shall be ignored.
	
	The elements of the input vectors are numbered from left to right across one or both of the vectors. For this purpose, the number of elements in a vector is given by **vec_step**(gentype_m_). The shuffle *mask* operand specifies, for each element of the result vector, which element of the one or two input vectors the result element gets.
	
	Examples:
	
	`uint4 mask = (uint4)(3, 2, 1, 0);`
	`float4 a;`
	`float4 r = shuffle(a, mask);`
	
	`uint8 mask = (uint8)(0, 1, 2, 3, 4, 5,`
	`6, 7);`
	`float4 a, b;`
	`float8 r = shuffle2(a, b, mask);`
	
	`uint4 mask;`
	`float8 a;`
	`float4 b;`
	
	`b = shuffle(a, mask);`
	
	Examples that are not valid are:
	
	`uint8 mask;`
	`short16 a;`
	`short8 b;`
	
	`b = shuffle(a, mask); //  not valid`

6.13.13. printf

The OpenCL C programming language implements the **printf** function.

Table 20. Built-in printf Function

Function	Description
int **printf**(constant char restrict *format*, ...)	The **printf** built-in function writes output to an implementation-defined stream such as stdout under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **printf** function returns when the end of the format string is encountered. **printf** returns 0 if it was executed successfully and -1 otherwise.

printf output synchronization

When the event that is associated with a particular kernel invocation is completed, the output of all printf() calls executed by this kernel invocation is flushed to the implementation-defined output stream. Calling **clFinish** on a command queue flushes all pending output by printf in previously enqueued and completed commands to the implementation-defined output stream. In the case that printf is executed from multiple work-items concurrently, there is no guarantee of ordering with respect to written data. For example, it is valid for the output of a work-item with a global id (0,0,1) to appear intermixed with the output of a work-item with a global id (0,0,4) and so on.

printf format string

The format shall be a character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream. The format is in the constant address space and must be resolvable at compile time, i.e. cannot be dynamically created by the executing program itself.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of a nonnegative decimal integer[51].

- An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of bytes to be written for **s** conversions. The precision takes the form of a period (**.**) followed by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional *vector specifier*.
- A *length modifier* that specifies the size of the argument. The *length modifier* is required with a vector specifier and together specifies the vector type. Implicit conversions between vector types are disallowed. If the *vector specifier* is not specified, the *length modifier* is optional.
- A *conversion specifier* character that specifies the type of conversion to be applied.

[51] Note that **0** is taken as a flag, not as the beginning of a field width.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)[52]

[52] The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and + flags both appear, the *space* flag is ignored.

The result is converted to an “alternative form”. For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are **not** removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

The vector specifier and its meaning is:

v*n* Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a vector argument, where *n* is the size of the vector and must be 2, 3, 4, 8 or 16.

The vector value is displayed in the following general form:

value1 C value2 C ... C value*n*

where C is a separator character. The value for this separator character is a comma.

If the vector specifier is not used, the length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `char` or `uchar` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `char` or `uchar` before printing).

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `short` or `ushort` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short` or `unsigned short` before printing).

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `long` or `ulong` argument. The **l** modifier is supported by the full profile. For the embedded profile, the **l** modifier is supported only if 64-bit integers are supported by the device.

If the vector specifier is used, the length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `charn` or `ucharn` argument (the argument will not be promoted).

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `shortn` or `ushortn` argument (the argument will not be promoted); that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a `halfn`[53] argument.

[53] Only if the cl_khr_fp16 extension is supported and has been enabled.

hl This modifier can only be used with the vector specifier. Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `intn` or `uintn` argument; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a `floatn` argument.

l(ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `longn` or `ulongn` argument; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a `doublen` argument. The **l** modifier is supported by the full profile. For the embedded profile, the **l** modifier is supported only if 64-bit integers or double-precision floating-point are supported by the device.

If a vector specifier appears without a length modifier, the behavior is undefined. The vector data type described by the vector specifier and length modifier must match the data type of the argument; otherwise the behavior is undefined.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d,i The `int`, `charn`, `shortn`, `intn` or `longn` argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o,u,

x,X The `unsigned int`, `ucharn`, `ushortn`, `uintn` or `ulongn` argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f,F A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A `double`, `halfn`, `floatn` or `doublen` argument representing an infinity is converted in one of the styles *[-]***inf** or *[-]***infinity** — which style is implementation-defined. A `double`, `halfn`, `floatn` or `doublen` argument representing a NaN is converted in one of the styles *[-]***nan** or *[-]***nan(***n-char-sequence***)** — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The **F** conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of **inf**, **infinity**, or **nan**, respectively[54].

[54] When applied to infinite and NaN values, the **-**, **+**, and *space* flag characters have their usual meaning; the # and **0** flag characters have no effect.

e,E A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted in the style *[-]d.ddd* **e±}***dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A `double`, `halfn`, `floatn` or `doublen` argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), depending on the value converted and the precision. Let *P* equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style **E** would have an exponent of *X*: — if *P* > *X* ≥ -4, the conversion is with style **f** (or **F**) and precision *P* **-** (*X* + 1). — otherwise, the conversion is with style **e** ***(or *E**) and precision *P* **-** 1. Finally, unless the # flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining. A `double`, `halfn`, `floatn` or `doublen` **e** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

a,A A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted in the style *[-]***0x***h.hhhh* **p±***d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[55] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing, then the precision is sufficient for an exact representation of the value; if the precision is zero and the # flag is not specified, no decimal point character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero. A `double`, `halfn`, `floatn` or `doublen` argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

[55] Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

> The conversion specifiers **e,E,g,G,a,A** convert a `float` or `half` argument that is a scalar type to a `double` only if the `double` data type is supported. If the `double` data type is not supported, the argument will be a `float` instead of a `double` and the `half` type will be converted to a `float`.

c The `int` argument is converted to an `unsigned char`, and the resulting character is written.

s The argument shall be a literal string[56]. Characters from the literal string array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

[56] No special provisions are made for multibyte characters. The behavior of **printf** with the **s** conversion specifier is undefined if the argument value is not a pointer to a literal string.

p The argument shall be a pointer to **void**. The pointer can refer to a memory region in the `global`, `constant`, `local`, `private`, or generic address space. The value of the pointer is converted to a sequence of printing characters in an implementation-defined manner.

% A **%** character is written. No argument is converted. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For **a** and **A** conversions, the value is correctly rounded to a hexadecimal floating number with the given precision.

A few examples of printf are given below:

```
float4  f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);

printf("f4 = %2.2v4hlf\n", f);
printf("uc = %#v4hhx\n", uc);
```

The above two printf calls print the following:

```
f4 = 1.00,2.00,3.00,4.00
uc = 0xfa,0xfb,0xfc,0xfd
```

A few examples of valid use cases of printf for the conversion specifier **s** are given below. The argument value must be a pointer to a literal string.

```
kernel void my_kernel( ... )
{
    printf("%s\n", "this is a test string\n");
}
```

A few examples of invalid use cases of printf for the conversion specifier **s** are given below:

```
kernel void my_kernel(global char *s, ... )
{
    printf("%s\n", s);
    constant char *p = "`this is a test string\n`";
    printf("%s\n", p);
    printf("%s\n", &p[3]);
}
```

A few examples of invalid use cases of printf where data types given by the vector specifier and length modifier do not match the argument type are given below:

```
kernel void my_kernel(global char *s, ... )
{
    uint2 ui = (uint2)(0x12345678, 0x87654321);

    printf("unsigned short value = (%#v2hx)\n", ui)
    printf("unsigned char value = (%#v2hhx)\n", ui)
}
```

Differences between OpenCL C and C99 printf

- The **l** modifier followed by a **c** conversion specifier or **s** conversion specifier is not supported by OpenCL C.

- The **ll**, **j**, **z**, **t**, and **L** length modifiers are not supported by OpenCL C but are reserved.
- The **n** conversion specifier is not supported by OpenCL C but is reserved.
- OpenCL C adds the optional *v*n vector specifier to support printing of vector types.
- The conversion specifiers **f**, **F**, **e**, **E**, **g**, **G**, **a**, **A** convert a `float` argument to a `double` only if the `double` data type is supported. Refer to the value of the `CL_DEVICE_DOUBLE_FP_CONFIG` device query. If the `double` data type is not supported, the argument will be a `float` instead of a `double`.
- For the embedded profile, the **l** length modifier is supported only if 64-bit integers are supported.
- In OpenCL C, **printf** returns 0 if it was executed successfully and -1 otherwise vs. C99 where **printf** returns the number of characters printed or a negative value if an output or encoding error occurred.
- In OpenCL C, the conversion specifier **s** can only be used for arguments that are literal strings.

6.13.14. Image Read and Write Functions

The built-in functions defined in this section can only be used with image memory objects. An image memory object can be accessed by specific function calls that read from and/or write to specific locations in the image.

Image memory objects that are being read by a kernel should be declared with the `read_only` qualifier. **write_image** calls to image memory objects declared with the read_only qualifier will generate a compilation error. Image memory objects that are being written to by a kernel should be declared with the write_only qualifier. **read_image** calls to image memory objects declared with the `write_only` qualifier will generate a compilation error. **read_image** and **write_image** calls to the same image memory object in a kernel are supported. Image memory objects that are being read and written by a kernel should be declared with the `read_write` qualifier.

The **read_image** calls returns a four component floating-point, integer or unsigned integer color value. The color values returned by **read_image** are identified as x, y, z, w where x refers to the red component, y refers to the green component, z refers to the blue component and w refers to the alpha component.

Samplers

The image read functions take a sampler argument. The sampler can be passed as an argument to the kernel using **clSetKernelArg**, or can be declared in the outermost scope of kernel functions, or it can be a constant variable of type `sampler_t` declared in the program source.

Sampler variables in a program are declared to be of type `sampler_t`. A variable of `sampler_t` type declared in the program source must be initialized with a 32-bit unsigned integer constant, which is interpreted as a bit-field specifiying the following properties:

- Addressing Mode
- Filter Mode
- Normalized Coordinates

These properties control how elements of an image object are read by **read_image{f|i|ui}**.

Samplers can also be declared as global constants in the program source using the following syntax.

```
const sampler_t <sampler name> = <value>
```

or

```
constant sampler_t <sampler name> = <value>
```

or

```
__constant sampler_t <sampler_name> = <value>
```

Note that samplers declared using the `constant` qualifier are not counted towards the maximum number of arguments pointing to the constant address space or the maximum size of the `constant` address space allowed per device (i.e. the value of the `CL_DEVICE_MAX_CONSTANT_ARGS` and `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE` device queries).

The sampler fields are described in the following table.

Table 21. Sampler Descriptor

Sampler State	**Description**
`<normalized coords>`	Specifies whether the *x*, *y* and *z* coordinates are passed in as normalized or unnormalized values. This must be a literal value and can be one of the following predefined enums: `CLK_NORMALIZED_COORDS_TRUE` or `CLK_NORMALIZED_COORDS_FALSE`. The samplers used with an image in multiple calls to **read_image{f|i|ui}** declared in a kernel must use the same value for <normalized coords>.

`<addressing mode>`	Specifies the image addressing mode, i.e. how out-of-range image coordinates are handled. This must be a literal value and can be one of the following predefined enums: `CLK_ADDRESS_MIRRORED_REPEAT` - Flip the image coordinate at every integer junction. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. `CLK_ADDRESS_REPEAT` - out-of-range image coordinates are wrapped to the valid range. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. `CLK_ADDRESS_CLAMP_TO_EDGE` - out-of-range image coordinates are clamped to the extent. `CLK_ADDRESS_CLAMP`[57] - out-of-range image coordinates will return a border color. `CLK_ADDRESS_NONE` - for this addressing mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined. For 1D and 2D image arrays, the addressing mode applies only to the *x* and (*x*, *y*) coordinates. The addressing mode for the coordinate which specifies the array index is always `CLK_ADDRESS_CLAMP_TO_EDGE`.
`<filter mode>`	Specifies the filter mode to use. This must be a literal value and can be one of the following predefined enums: `CLK_FILTER_NEAREST` or `CLK_FILTER_LINEAR`. Refer to the detailed description of these filter modes.

[57] This is similar to the `GL_ADDRESS_CLAMP_TO_BORDER` addressing mode.

Examples:

```
const sampler_t samplerA = CLK_NORMALIZED_COORDS_TRUE |
                           CLK_ADDRESS_REPEAT |
                           CLK_FILTER_NEAREST;
```

`samplerA` specifies a sampler that uses normalized coordinates, the repeat addressing mode and a nearest filter.

The maximum number of samplers that can be declared in a kernel can be queried using the `CL_DEVICE_MAX_SAMPLERS` token in **clGetDeviceInfo**.

Determining the border color or value

If `<addressing mode>` in sampler is `CLK_ADDRESS_CLAMP`, then out-of-range image coordinates return the border color. The border color selected depends on the image channel order and can be one of the following values:

- If the image channel order is `CL_A`, `CL_INTENSITY`, `CL_Rx`, `CL_RA`, `CL_RGx`, `CL_RGBx`, `CL_sRGBx`, `CL_ARGB`, `CL_BGRA`, `CL_ABGR`, `CL_RGBA`, `CL_sRGBA` or `CL_sBGRA`, the border color is `(0.0f, 0.0f, 0.0f, 0.0f)`.
- If the image channel order is `CL_R`, `CL_RG`, `CL_RGB`, or `CL_LUMINANCE`, the border color is `(0.0f, 0.0f, 0.0f, 1.0f)`.
- If the image channel order is `CL_DEPTH`, the border value is `0.0f`.

sRGB Images

The built-in image read functions will perform sRGB to linear RGB conversions if the image is an sRGB image. Writing to sRGB images from a kernel is an optional extension. The **cl_khr_srgb_image_writes** extension will be reported in the `CL_DEVICE_EXTENSIONS` string if a device supports writing to sRGB images using **write_imagef**. **clGetSupportedImageFormats** will return the supported sRGB images if `CL_MEM_READ_WRITE` or `CL_MEM_WRITE_ONLY` is specified in *flags* argument and the device supports writing to an sRGB image. If **cl_khr_srgb_image_writes** is supported, the built-in image write functions will perform the linear to sRGB conversion.

Only the R, G and B components are converted from linear to sRGB and vice-versa. The alpha component is returned as is.

Built-in Image Read Functions

The following built-in function calls to read images with a sampler[58] are supported.

[58] The built-in function calls to read images with a sampler are not supported for `image1d_buffer_t` image types.

Table 22. Built-in Image Read Functions

Function	Description

float4 **read_imagef**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

int4 **read_imagei**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*) uint4 **read_imageui**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) uint4 **read_imageui**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

float4 **read_imagef**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) float4 **read_imagef**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description are undefined.

int4 **read_imagei**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) int4 **read_imagei**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*) uint4 **read_imageui**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) uint4 **read_imageui**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

float4 **read_imagef**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) float4 **read_imagef**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with image_channel_data_type set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with image_channel_data_type set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with image_channel_data_type set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with image_channel_data_type values not specified in the description above are undefined.

int4 **read_imagei**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) int4 **read_imagei**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*) uint4 **read_imageui**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) uint4 **read_imageui**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

float4 **read_imagef**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) float4 **read_imagef**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*)	Use *coord* to do an element lookup in the 1D image object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

int4 **read_imagei**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) int4 **read_imagei**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*) uint4 **read_imageui**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) uint4 **read_imageui**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*)	Use *coord* to do an element lookup in the 1D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

float4 **read_imagef**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with image_channel_data_type set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with image_channel_data_type set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with image_channel_data_type set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with image_channel_data_type values not specified in the description above are undefined.

int4 **read_imagei**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*) uint4 **read_imageui**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) uint4 **read_imageui**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

float **read_imagef**(read_only image2d_depth_t *image*, sampler_t *sampler*, int2 *coord*) float **read_imagef**(read_only image2d_depth_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D depth image object specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for depth image objects with *image_channel_data_type* values not specified in the description above are undefined.
float **read_imagef**(read_only image2d_array_depth_t *image*, sampler_t *sampler*, int4 *coord*) float **read_imagef**(read_only image2d_array_depth_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Built-in Image Sampler-less Read Functions

The sampler-less image read functions behave exactly as the corresponding built-in image read functions that take integer coordinates and a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode to `CLK_ADDRESS_NONE`. There is one exception when the *image_channel_data_type* is a floating point type (such as `CL_FLOAT`). In this exceptional case, when channel data values are denormalized, the sampler-less image read function may return the denormalized data, while the image read function with a sampler argument may flush the denormalized channel data values to zero.

aQual in the following table refers to one of the access qualifiers. For samplerless read functions this may be `read_only` or `read_write`.

Table 23. Built-in Image Sampler-less Read Functions

Function	Description
float4 **read_imagef**(*aQual* image2d_t *image*, int2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

int4 **read_imagei**(*aQual* image2d_t *image*, int2 *coord*) uint4 **read_imageui**(*aQual* image2d_t *image*, int2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

float4 **read_imagef**(*aQual* image3d_t *image*, int4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description are undefined.

int4 **read_imagei**(*aQual* image3d_t *image*, int4 *coord*) uint4 **read_imageui**(*aQual* image3d_t *image*, int4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

float4 **read_imagef**(*aQual* image2d_array_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

int4 **read_imagei**(*aQual* image2d_array_t *image*, int4 *coord*) uint4 **read_imageui**(*aQual* image2d_array_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

float4 **read_imagef**(*aQual* image1d_t *image*, int *coord*) float4 **read_imagef**(*aQual* image1d_buffer_t *image*, int *coord*)	Use *coord* to do an element lookup in the 1D image or 1D image buffer object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

int4 **read_imagei**(*aQual* image1d_t *image*, int *coord*) uint4 **read_imageui**(*aQual* image1d_t *image*, int *coord*) int4 **read_imagei**(*aQual* image1d_buffer_t *image*, int *coord*) uint4 **read_imageui**(*aQual* image1d_buffer_t *image*, int *coord*)	Use *coord* to do an element lookup in the 1D image or 1D image buffer object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

float4 **read_imagef**(*aQual* image1d_array_t *image*, int2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

int4 **read_imagei**(*aQual* image1d_array_t *image*, int2 *coord*) uint4 **read_imageui**(*aQual* image1d_array_t *image*, int2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.
float **read_imagef**(*aQual* image2d_depth_t *image*, int2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D depth image object specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

float **read_imagef**(*aQual* image2d_array_depth_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Built-in Image Write Functions

The following built-in function calls to write images are supported. Note that image writes to sRGB images are only supported if the **cl_khr_srgb_image_writes** extension is supported; otherwise the behavior of writing to a sRGB image is undefined.

aQual in the following table refers to one of the access qualifiers. For write functions this may be `write_only` or `read_write`.

Table 24. Built-in Image Write Functions

Function	Description

void **write_imagef**(*aQual* image2d_t *image*, int2 *coord*, float4 *color*) void **write_imagei**(*aQual* image2d_t *image*, int2 *coord*, int4 *color*) void **write_imageui**(*aQual* image2d_t *image*, int2 *coord*, uint4 *color*)	Write *color* value to location specified by *coord.xy* in the 2D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x* and *coord.y* are considered to be unnormalized coordinates, and must be in the range [0, image width-1] and [0, image height-1] respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with *x* and *y* coordinate values that are not in the range [0, image width-1] and [0, image height-1], respectively, is undefined.

void **write_imagef**(*aQual* image2d_array_t *image*, int4 *coord*, float4 *color*) void **write_imagei**(*aQual* image2d_array_t *image*, int4 *coord*, int4 *color*) void **write_imageui**(*aQual* image2d_array_t *image*, int4 *coord*, uint4 *color*)	Write *color* value to location specified by *coord.xy* in the 2D image identified by *coord.z* in the 2D image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates, and must be in the range [0, image width-1] and [0, image height-1], and [0, image number of layers-1], respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range [0, image width-1], [0, image height-1], and [0, image number of layers-1], respectively, is undefined.

void **write_imagef**(*aQual* image1d_t *image*, int *coord*, float4 *color*) void **write_imagei**(*aQual* image1d_t *image*, int *coord*, int4 *color*) void **write_imageui**(*aQual* image1d_t *image*, int *coord*, uint4 *color*) void **write_imagef**(*aQual* image1d_buffer_t *image*, int *coord*, float4 *color*) void **write_imagei**(*aQual* image1d_buffer_t *image*, int *coord*, int4 *color*) void **write_imageui**(*aQual* image1d_buffer_t *image*, int *coord*, uint4 *color*)	Write *color* value to location specified by *coord* in the 1D image or 1D image buffer object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord* is considered to be an unnormalized coordinate, and must be in the range [0, image width-1]. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above, or with a coordinate value that is not in the range [0, image width-1], is undefined.

void **write_imagef**(*aQual* image1d_array_t *image*, int2 *coord*, float4 *color*) void **write_imagei**(*aQual* image1d_array_t *image*, int2 *coord*, int4 *color*) void **write_imageui**(*aQual* image1d_array_t *image*, int2 *coord*, uint4 *color*)	Write *color* value to location specified by *coord.x* in the 1D image identified by *coord.y* in the 1D image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x* and *coord.y* are considered to be unnormalized coordinates and must be in the range [0, image width-1] and [0, image number of layers-1], respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*) coordinate values that are not in the range [0, image width-1] and [0, image number of layers-1], respectively, is undefined.

void **write_imagef**(*aQual* image2d_depth_t *image*, int2 *coord*, float *depth*)	Write *depth* value to location specified by *coord.xy* in the 2D depth image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the depth value. *coord.x* and *coord.y* are considered to be unnormalized coordinates, and must be in the range [0, image width-1], and [0, image height-1], respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to `CL_UNORM_INT16`, `CL_UNORM_INT24` or `CL_FLOAT`. Appropriate data format conversion will be done to convert depth valye from a floating-point value to actual data format associated with the image. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*) coordinate values that are not in the range [0, image width-1] and [0, image height-1], respectively, is undefined.
void **write_imagef**(*aQual* image2d_array_depth_t *image*, int4 *coord*, float *depth*)	Write *depth* value to location specified by *coord.xy* in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the depth value. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates, and must be in the range [0, image width-1], [0, image height-1], and [0, image number of layers-1], respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to `CL_UNORM_INT16`, `CL_UNORM_INT24` or `CL_FLOAT`. Appropriate data format conversion will be done to convert depth valye from a floating-point value to actual data format associated with the image. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range [0, image width-1], [0, image height-1], [0, image number of layers-1], respectively, is undefined.

void **write_imagef**(*aQual* image3d_t *image*, int4 *coord*, float4 *color*) void **write_imagei**(*aQual* image3d_t *image*, int4 *coord*, int4 *color*) void **write_imageui**(*aQual* image3d_t *image*, int4 *coord*, uint4 *color*)	Write color value to location specified by *coord.xyz* in the 3D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates, and must be in the range [0, image width-1], [0, image height-1], and [0, image depth-1], respectively. **write_imagef** can only be used with image objects created with image_channel_data_type set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with image_channel_data_type set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with image_channel_data_type set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range [0, image width-1], [0, image height-1], and [0, image depth-1], respectively, is undefined.

Built-in Image Query Functions

The following built-in function calls to query image information are supported.

aQual in the following table refers to one of the access qualifiers. For query functions this may be `read_only`, `write_only` or `read_write`.

Table 25. Built-in Image Query Functions

Function	Description
int **get_image_width**(*aQual* image1d_t *image*) int **get_image_width**(*aQual* image1d_buffer_t *image*) int **get_image_width**(*aQual* image2d_t *image*) int **get_image_width**(*aQual* image3d_t *image*) int **get_image_width**(*aQual* image1d_array_t *image*) int **get_image_width**(*aQual* image2d_array_t *image*) int **get_image_width**(*aQual* image2d_depth_t *image*) int **get_image_width**(*aQual* image2d_array_depth_t *image*)	Return the image width in pixels.
int **get_image_height**(*aQual* image2d_t *image*) int **get_image_height**(*aQual* image3d_t *image*) int **get_image_height**(*aQual* image2d_array_t *image*) int **get_image_height**(*aQual* image2d_depth_t *image*) int **get_image_height**(*aQual* image2d_array_depth_t *image*)	Return the image height in pixels.
int **get_image_depth**(image3d_t *image*)	Return the image depth in pixels.

int **get_image_channel_data_type**(*aQual* image1d_t *image*)
int **get_image_channel_data_type**(*aQual* image1d_buffer_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_t *image*)
int **get_image_channel_data_type**(*aQual* image3d_t *image*)
int **get_image_channel_data_type**(*aQual* image1d_array_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_array_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_depth_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_array_depth_t *image*)
	Return the channel data type. Valid values are:
	
	`CLK_SNORM_INT8`
	`CLK_SNORM_INT16`
	`CLK_UNORM_INT8`
	`CLK_UNORM_INT16`
	`CLK_UNORM_SHORT_565`
	`CLK_UNORM_SHORT_555`
	`CLK_UNORM_SHORT_101010`
	`CLK_SIGNED_INT8`
	`CLK_SIGNED_INT16`
	`CLK_SIGNED_INT32`
	`CLK_UNSIGNED_INT8`
	`CLK_UNSIGNED_INT16`
	`CLK_UNSIGNED_INT32`
	`CLK_HALF_FLOAT`
	`CLK_FLOAT`

int **get_image_channel_order**(*aQual* image1d_t *image*)
int **get_image_channel_order**(*aQual* image1d_buffer_t *image*)
int **get_image_channel_order**(*aQual* image2d_t *image*)
int **get_image_channel_order**(*aQual* image3d_t *image*)
int **get_image_channel_order**(*aQual* image1d_array_t *image*)
int **get_image_channel_order**(*aQual* image2d_array_t *image*)
int **get_image_channel_order**(*aQual* image2d_depth_t *image*)
int **get_image_channel_order**(*aQual* image2d_array_depth_t *image*)
	Return the image channel order. Valid values are:
	
	`CLK_A`
	`CLK_R`
	`CLK_Rx`
	`CLK_RG`
	`CLK_RGx`
	`CLK_RA`
	`CLK_RGB`
	`CLK_RGBx`
	`CLK_RGBA`
	`CLK_ARGB`
	`CLK_BGRA`
	`CLK_INTENSITY`
	`CLK_LUMINANCE`
	`CLK_ABGR`
	`CLK_DEPTH`
	`CLK_sRGB`
	`CLK_sRGBx`
	`CLK_sRGBA`
	`CLK_sBGRA`

int2 **get_image_dim**(*aQual* image2d_t *image*) int2 **get_image_dim**(*aQual* image2d_array_t *image*) int2 **get_image_dim**(*aQual* image2d_depth_t *image*) int2 **get_image_dim**(*aQual* image2d_array_depth_t *image*)	Return the 2D image width and height as an int2 type. The width is returned in the *x* component, and the height in the *y* component.
int4 **get_image_dim**(*aQual* image3d_t *image*)	Return the 3D image width, height, and depth as an `int4` type. The width is returned in the *x* component, height in the *y* component, depth in the *z* component and the *w* component is 0.
size_t **get_image_array_size**(*aQual* image2d_array_t *image*) size_t **get_image_array_size**(*aQual* image2d_array_depth_t *image*)	Return the number of images in the 2D image array.
size_t **get_image_array_size**(*aQual* image1d_array_t *image*)	Return the number of images in the 1D image array.

The values returned by **get_image_channel_data_type** and **get_image_channel_order** as specified in Built-in Image Query Functions with the `CLK_` prefixes correspond to the `CL_` prefixes used to describe the image channel order and data type in the OpenCL Specification. For example, both `CL_UNORM_INT8` and `CLK_UNORM_INT8` refer to an image channel data type that is an unnormalized unsigned 8-bit integer.

Reading and writing to the same image in a kernel

The **atomic_work_item_fence**(`CLK_IMAGE_MEM_FENCE`) built-in function can be used to make sure that sampler-less writes are visible to later reads by the same work-item. Only a scope of `memory_scope_work_item` and an order of `memory_order_acq_rel` is valid for `atomic_work_item_fence` when passed the `CLK_IMAGE_MEM_FENCE` flag. If multiple work-items are writing to and reading from multiple locations in an image, the **work_group_barrier**(`CLK_IMAGE_MEM_FENCE`) should be used.

Consider the following example:

```
kernel void
foo(read_write image2d_t img, ... )
{
    int2 coord;
    coord.x = (int)get_global_id(0);
    coord.y = (int)get_global_id(1);

    float4 clr = read_imagef(img, coord);
    ...
    write_imagef(img, coord, clr);

    // required to ensure that following read from image at
    // location coord returns the latest color value.
    atomic_work_item_fence(
        `CLK_IMAGE_MEM_FENCE`,
        memory_order_acq_rel,
        memory_scope_work_item);

    float4 clr_new = read_imagef(img, coord);
    ...

}
```

Mapping image channels to color values returned by read_image and color values passed to write_image to image channels

The following table describes the mapping of the number of channels of an image element to the appropriate components in the `float4`, `int4` or `uint4` vector data type for the color values returned by **read_image{f|i|ui}** or supplied to **write_image{f|i|ui}**. The unmapped components will be set to 0.0 for red, green and blue channels and will be set to 1.0 for the alpha channel.

Channel Order	`float4`, `int4` or `uint4` components of channel data
`CL_R`, `CL_Rx`	(r, 0.0, 0.0, 1.0)
`CL_A`	(0.0, 0.0, 0.0, a)
`CL_RG`, `CL_RGx`	(r, g, 0.0, 1.0)
`CL_RA`	(r, 0.0, 0.0, a)
`CL_RGB`, `CL_RGBx`, `CL_sRGB`, `CL_sRGBx`	(r, g, b, 1.0)
`CL_RGBA`, `CL_BGRA`, `CL_ARGB`, `CL_ABGR`, `CL_sRGBA`, `CL_sBGRA`	(r, g, b, a)

`CL_INTENSITY`	(I, I, I, I)
`CL_LUMINANCE`	(L, L, L, 1.0)

For `CL_DEPTH` images, a scalar value is returned by **read_imagef** or supplied to **write_imagef**.

> A kernel that uses a sampler with the `CL_ADDRESS_CLAMP` addressing mode with multiple images may result in additional samplers being used internally by an implementation. If the same sampler is used with multiple images called via **read_image{f|i|ui}**, then it is possible that an implementation may need to allocate an additional sampler to handle the different border color values that may be needed depending on the image formats being used. These implementation allocated samplers will count against the maximum sampler values supported by the device and given by `CL_DEVICE_MAX_SAMPLERS`. Enqueuing a kernel that requires more samplers than the implementation can support will result in a `CL_OUT_OF_RESOURCES` error being returned.

6.13.15. Work-group Functions

The OpenCL C programming language implements the following built-in functions that operate on a work-group level. These built-in functions must be encountered by all work-items in a work-group executing the kernel. We use the generic type name `gentype` to indicate the built-in data types `half`[59], `int`, `uint`, `long`, `ulong`, `float` or `double`[60] as the type for the arguments.

[59] Only if the **cl_khr_fp16** extension is supported and has been enabled.

[60] Only if double precision is supported.

Table 26. Built-in Work-group Functions

Function	**Description**
int **work_group_all**(int *predicate*)	Evaluates *predicate* for all work-items in the work-group and returns a non-zero value if *predicate* evaluates to non-zero for all work-items in the work-group.
int **work_group_any**(int *predicate*)	Evaluates *predicate* for all work-items in the work-group and returns a non-zero value if *predicate* evaluates to non-zero for any work-items in the work-group.
gentype **work_group_broadcast**(gentype *a*, size_t *local_id*) gentype **work_group_broadcast**(gentype *a*, size_t *local_idx, size_t _local_id_y)* *gentype **work_group_broadcast**(gentype a, size_t local_idx, size_t _local_id__y, size_t _local_id_z*)	Broadcast the value of *x* for work-item identified by *local_id* to all work-items in the work-group. *local_id* must be the same value for all work-items in the work-group.
gentype **work_group_reduce_<op>**(gentype *x*)	Return result of reduction operation specified by **<op>** for all values of *x* specified by work-items in a work-group.

gentype **work_group_scan_exclusive_<op>**(gentype *x*)	Do an exclusive scan operation specified by **<op>** of all values specified by work-items in the work-group. The scan results are returned for each work-item. The scan order is defined by increasing 1D linear global ID within the work-group.
gentype **work_group_scan_inclusive_<op>**(gentype *x*)	Do an inclusive scan operation specified by **<op>** of all values specified by work-items in the work-group. The scan results are returned for each work-item. The scan order is defined by increasing 1D linear global ID within the work-group.

The **<op>** in **work_group_reduce_<op>**, **work_group_scan_exclusive_<op>** and **work_group_scan_inclusive_<op>** defines the operator and can be **add**, **min** or **max**.

The inclusive scan operation takes a binary operator **op** with an identity I and n (where n is the size of the work-group) elements [a_0, a_1, … a_{n-1}] and returns [a_0, (a_0 **op** a_1), … (a_0 **op** a_1 **op** … **op** a_{n-1})]. If **op** = add, the identity I is 0. If **op** = min, the identity I is `INT_MAX`, `UINT_MAX`, `LONG_MAX`, `ULONG_MAX`, for `int`, `uint`, `long`, `ulong` types and is `+INF` for floating-point types. Similarly if **op** = max, the identity I is `INT_MIN`, 0, `LONG_MIN`, 0 and `-INF`.

Consider the following example:

```
void foo(int *p)
{
    ...
    int prefix_sum_val = work_group_scan_inclusive_add(
                            p[get_local_id(0)]);
}
```

For the example above, let's assume that the work-group size is 8 and *p* points to the following elements [3 1 7 0 4 1 6 3]. Work-item 0 calls **work_group_scan_inclusive_add** with 3 and returns 3. Work-item 1 calls **work_group_scan_inclusive_add** with 1 and returns 4. The full set of values returned by **work_group_scan_inclusive_add** for work-items 0 … 7 are [3 4 11 11 15 16 22 25].

The exclusive scan operation takes a binary associative operator **op** with an identity I and n (where n is the size of the work-group) elements [a_0, a_1, … a_{n-1}] and returns [I, a_0, (a_0 **op** a_1), … (a_0 **op** a_1 **op** … **op** a_{n-2})]. For the example above, the exclusive scan add operation on the ordered set [3 1 7 0 4 1 6 3] would return [0 3 4 11 11 15 16 22].

The order of floating-point operations is not guaranteed for the **work_group_reduce_<op>**, **work_group_scan_inclusive_<op>** and **work_group_scan_exclusive_<op>** built-in functions that operate on `half`, `float` and `double` data types. The order of these floating-point operations is also non-deterministic for a given work-group.

6.13.16. Pipe Functions

A pipe is identified by specifying the `pipe` keyword with a type. The data type specifies the size of each packet in the pipe. The `pipe` keyword is a type modifier. When it is applied to another type **T**, the result is a pipe type whose elements (or packets) are of type **T**. The packet type **T** may be any supported OpenCL C scalar and vector integer or floating-point data types, or a user-defined type built from these scalar and vector data types.

Examples:

```
pipe int4 pipeA; // a pipe with int4 packets

pipe user_type_t pipeB; // a pipe with user_type_t packets
```

The `read_only` (or `__read_only`) and `write_only` (or `__write_only`) qualifiers must be used with the `pipe` qualifier when a pipe is a parameter of a kernel or of a user-defined function to identify if a pipe can be read from or written to by a kernel and its callees and enqueued child kernels. If no qualifier is specified, `read_only` is assumed.

A kernel cannot read from and write to the same pipe object. Using the `read_write` (or `__read_write`) qualifier with the `pipe` qualifier is a compilation error.

In the following example

```
kernel void
foo (read_only pipe fooA_t pipeA,
     write_only pipe fooB_t pipeB)
{
    ...
}
```

`pipeA` is a read-only pipe object, and `pipeB` is a write-only pipe object.

The macro `CLK_NULL_RESERVE_ID` refers to an invalid reservation ID.

Restrictions

- Pipes can only be passed as arguments to a function (including kernel functions). The C operators cannot be used with variables declared with the pipe qualifier.
- The `pipe` qualifier cannot be used with variables declared inside a kernel, a structure or union field, a pointer type, an array, global variables declared in program scope or the return type of a function.

Built-in Pipe Read and Write Functions

The OpenCL C programming language implements the following built-in functions that read from or write to a pipe. We use the generic type name `gentype` to indicate the built-in OpenCL C scalar or vector integer or floating-point data types[61] or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in the following table.

[61] The `half` scalar and vector types can only be used if the **cl_khr_fp16** extension is supported and has been enabled. The `double` scalar and vector types can only be used if `double` precision is supported.

Table 27. Built-in Pipe Functions

Function	Description
int **read_pipe**(read_only pipe gentype *p*, gentype **ptr*)	Read packet from pipe *p* into *ptr*. Returns 0 if **read_pipe** is successful and a negative value if the pipe is empty.
int **write_pipe**(write_only pipe gentype *p*, const gentype **ptr*)	Write packet specified by *ptr* to pipe *p*. Returns 0 if **write_pipe** is successful and a negative value if the pipe is full.
int **read_pipe**(read_only pipe gentype *p*, reserve_id_t *reserve_id*, uint *index*, gentype **ptr*)	Read packet from the reserved area of the pipe referred to by *reserve_id* and *index* into *ptr*. The reserved pipe entries are referred to by indices that go from 0 … *num_packets* - 1. Returns 0 if **read_pipe** is successful and a negative value otherwise.
int **write_pipe**(write_only pipe gentype *p*, reserve_id_t *reserve_id*, uint *index*, const gentype **ptr*)	Write packet specified by *ptr* to the reserved area of the pipe referred to by *reserve_id* and *index*. The reserved pipe entries are referred to by indices that go from 0 … *num_packets* - 1. Returns 0 if **write_pipe** is successful and a negative value otherwise.
reserve_id_t **reserve_read_pipe**(read_only pipe gentype *p*, uint *num_packets*) reserve_id_t **reserve_write_pipe**(write_only pipe gentype *p*, uint *num_packets*)	Reserve *num_packets* entries for reading from or writing to pipe *p*. Returns a valid reservation ID if the reservation is successful.
void **commit_read_pipe**(read_only pipe gentype *p*, reserve_id_t *reserve_id*) void **commit_write_pipe**(write_only pipe gentype *p*, reserve_id_t *reserve_id*)	Indicates that all reads and writes to *num_packets* associated with reservation *reserve_id* are completed.
bool **is_valid_reserve_id**(reserve_id_t *reserve_id*)	Return *true* if *reserve_id* is a valid reservation ID and *false* otherwise.

Built-in Work-group Pipe Read and Write Functions

The OpenCL C programming language implements the following built-in pipe functions that operate at a work-group level. These built-in functions must be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the behavior is undefined.

We use the generic type name `gentype` to indicate the built-in OpenCL C scalar or vector integer or floating-point data types[62] or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in the following table.

[62] The `half` scalar and vector types can only be used if the **cl_khr_fp16** extension is supported and has been enabled. The `double` scalar and vector types can only be used if `double` precision is supported.

Table 28. Built-in Pipe Work-group Functions

Function	Description
reserve_id_t **work_group_reserve_read_pipe**(read_only pipe gentype *p*, uint *num_packets*) reserve_id_t **work_group_reserve_write_pipe**(write_only pipe gentype *p*, uint *num_packets*)	Reserve *num_packets* entries for reading from or writing to pipe *p*. Returns a valid reservation ID if the reservation is successful. The reserved pipe entries are referred to by indices that go from 0 … *num_packets* - 1.
void **work_group_commit_read_pipe**(read_only pipe gentype *p*, reserve_id_t *reserve_id*) void **work_group_commit_write_pipe**(write_only pipe gentype *p*, reserve_id_t *reserve_id*)	Indicates that all reads and writes to *num_packets* associated with reservation *reserve_id* are completed.

> The **read_pipe** and **write_pipe** functions that take a reservation ID as an argument can be used to read from or write to a packet index. These built-ins can be used to read from or write to a packet index one or multiple times. If a packet index that is reserved for writing is not written to using the **write_pipe** function, the contents of that packet in the pipe are undefined. **commit_read_pipe** and **work_group_commit_read_pipe** remove the entries reserved for reading from the pipe. **commit_write_pipe** and **work_group_commit_write_pipe** ensures that the entries reserved for writing are all added in-order as one contiguous set of packets to the pipe.

There can only be the value of the `CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS` device query reservations active (i.e. reservation IDs that have been reserved but not committed) per work-item or work-group for a pipe in a kernel executing on a device.

Work-item based reservations made by a work-item are ordered in the pipe as they are ordered in the program. Reservations made by different work-items that belong to the same work-group can be ordered using the work-group barrier function. The order of work-item based reservations that belong to different work-groups is implementation defined.

Work-group based reservations made by a work-group are ordered in the pipe as they are ordered in the program. The order of work-group based reservations by different work-groups is implementation defined.

Built-in Pipe Query Functions

The OpenCL C programming language implements the following built-in query functions for a pipe. We use the generic type name `gentype` to indicate the built-in OpenCL C scalar or vector integer or floating-point data types[63] or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in the following table.

[63] The `half` scalar and vector types can only be used if the **cl_khr_fp16** extension is supported and has been enabled. The `double` scalar and vector types can only be used if `double` precision is supported.

aQual in the following table refers to one of the access qualifiers. For pipe query functions this may be `read_only` or `write_only`.

Table 29. Built-in Pipe Query Functions

Function	Description
uint **get_pipe_num_packets**(*aQual* pipe gentype *p*)	Returns the number of available entries in the pipe. The number of available entries in a pipe is a dynamic value. The value returned should be considered immediately stale.
uint **get_pipe_max_packets**(*aQual* pipe gentype *p*)	Returns the maximum number of packets specified when *pipe* was created.

Restrictions

The following behavior is undefined:

- A kernel fails to call **reserve_pipe** before calling **read_pipe** or **write_pipe** that take a reservation ID.
- A kernel calls **read_pipe**, **write_pipe**, **commit_read_pipe** or **commit_write_pipe** with an invalid reservation ID.
- A kernel calls **read_pipe** or **write_pipe** with an valid reservation ID but with an *index* that is not a value in the range [0, *num_packets*-1] specified to the corresponding call to *reserve_pipe*.
- A kernel calls **read_pipe** or **write_pipe** with a reservation ID that has already been committed (i.e. a **commit_read_pipe** or **commit_write_pipe** with this reservation ID has already been called).
- A kernel fails to call **commit_read_pipe** for any reservation ID obtained by a prior call to **reserve_read_pipe**.
- A kernel fails to call **commit_write_pipe** for any reservation ID obtained by a prior call to **reserve_write_pipe**.
- The contents of the reserved data packets in the pipe are undefined if the kernel does not call **write_pipe** for all entries that were reserved by the corresponding call to **reserve_pipe**.
- Calls to **read_pipe** that takes a reservation ID and **commit_read_pipe** or **write_pipe** that takes a reservation ID and **commit_write_pipe** for a given reservation ID must be called by the same kernel that made the reservation using **reserve_read_pipe** or **reserve_write_pipe**. The reservation ID cannot be passed to another kernel including child kernels.

6.13.17. Enqueuing Kernels

OpenCL 2.0 allows a kernel to independently enqueue to the same device, without host interaction.

A kernel may enqueue code represented by Block syntax, and control execution order with event dependencies including user events and markers. There are several advantages to using the Block syntax: it is more compact; it does not require a cl_kernel object; and enqueuing can be done as a single semantic step.

The following table describes the list of built-in functions that can be used to enqueue a kernel(s).

The macro `CLK_NULL_EVENT` refers to an invalid device event. The macro `CLK_NULL_QUEUE` refers to an invalid device queue.

Built-in Functions - Enqueuing a kernel

Table 30. Built-in Kernel Enqueue Functions

Built-in Function	Description
int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, void (^*block*)(void)) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*, void (^*block*)(void)) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, void (^*block*)(local void *, ...), uint size0, ...) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*, void (^*block*)(local void *, ...), uint size0, ...)	Enqueue the block for execution to *queue*. If an event is returned, **enqueue_kernel** performs an implicit retain on the returned event.

The **enqueue_kernel** built-in function allows a work-item to enqueue a block. Work-items can enqueue multiple blocks to a device queue(s).

The **enqueue_kernel** built-in function returns `CLK_SUCCESS` if the block is enqueued successfully and returns `CLK_ENQUEUE_FAILURE` otherwise. If the -g compile option is specified in compiler options passed to **clCompileProgram** or **clBuildProgram** when compiling or building the parent program, the following errors may be returned instead of `CLK_ENQUEUE_FAILURE` to indicate why **enqueue_kernel** failed to enqueue the block:

- `CLK_INVALID_QUEUE` if *queue* is not a valid device queue.
- `CLK_INVALID_NDRANGE` if *ndrange* is not a valid ND-range descriptor or if the program was compiled with `-cl-uniform-work-group-size` and the *local_work_size* is specified in *ndrange* but the *global_work_size* specified in *ndrange* is not a multiple of the *local_work_size*.
- `CLK_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- `CLK_DEVICE_QUEUE_FULL` if *queue* is full.
- `CLK_INVALID_ARG_SIZE` if size of local memory arguments is 0.
- `CLK_EVENT_ALLOCATION_FAILURE` if *event_ret* is not `NULL` and an event could not be allocated.
- `CLK_OUT_OF_RESOURCES` if there is a failure to queue the block in *queue* because of insufficient resources needed to execute the kernel.

Below are some examples of how to enqueue a block.

```
kernel void
my_func_A(global int *a, global int *b, global int *c)
{
    ...
}

kernel void
my_func_B(global int *a, global int *b, global int *c)
{
    ndrange_t ndrange;
    // build ndrange information
    ...

    // example - enqueue a kernel as a block
    enqueue_kernel(get_default_queue(), ndrange,
                   ^{my_func_A(a, b, c);});


    ...
}

kernel void
my_func_C(global int *a, global int *b, global int *c)
{
    ndrange_t ndrange;
    // build ndrange information
    ...

    // note that a, b and c are variables in scope of
    // the block
    void (^my_block_A)(void) = ^{my_func_A(a, b, c);};

    // enqueue the block variable
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   my_block_A);
    ...
}
```

The example below shows how to declare a block literal and enqueue it.

```
kernel void
my_func(global int *a, global int *b)
{
    ndrange_t ndrange;
    // build ndrange information
    ...

    // note that a, b and c are variables in scope of
    // the block
    void (^my_block_A)(void) =
    ^{
        size_t id = get_global_id(0);
        b[id] += a[id];
    };

    // enqueue the block variable
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   my_block_A);

    // or we could have done the following
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   ^{
                       size_t id = get_global_id(0);
                       b[id] += a[id];
                   };
}
```

Blocks passed to enqueue_kernel cannot use global variables or stack variables local to the enclosing lexical scope that are a pointer type in the `local` or `private` address space.

Example:

```
kernel void
foo(global int *a, local int *lptr, ...)
{
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   ^{
                       size_t id = get_global_id(0);
                       local int *p = lptr; // undefined behavior
                   } );
}
```

Arguments that are a pointer type to local address space

A block passed to enqueue_kernel can have arguments declared to be a pointer to `local` memory. The enqueue_kernel built-in function variants allow blocks to be enqueued with a variable number of arguments. Each argument must be declared to be a `void` pointer to local memory. These enqueue_kernel built-in function variants also have a corresponding number of arguments each of type `uint` that follow the block argument. These arguments specify the size of each local memory pointer argument of the enqueued block.

Some examples follow:

```
kernel void
my_func_A_local_arg1(global int *a, local int *lptr, ...)
{
    ...
}

kernel void
my_func_A_local_arg2(global int *a,
                     local int *lptr1, local float4 *lptr2, ...)
{
    ...
}

kernel void
my_func_B(global int *a, ...)
{
    ...

    ndrange_t ndrange = ndrange_1d(...);

    uint local_mem_size = compute_local_mem_size();

    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   ^(local void *p){
                       my_func_A_local_arg1(a, (local int *)p, ...);},
                   local_mem_size);
}

kernel void
my_func_C(global int *a, ...)
{
    ...
    ndrange_t ndrange = ndrange_1d(...);

    void (^my_blk_A)(local void *, local void *) =
        ^(local void *lptr1, local void *lptr2){
        my_func_A_local_arg2(
```

```
            a,
            (local int *)lptr1,
            (local float4 *)lptr2, ...);};

    // calculate local memory size for lptr
    // argument in local address space for my_blk_A
    uint local_mem_size = compute_local_mem_size();

    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   my_blk_A,
                   local_mem_size, local_mem_size*4);
}
```

A Complete Example

The example below shows how to implement an iterative algorithm where the host enqueues the first instance of the nd-range kernel (dp_func_A). The kernel dp_func_A will launch a kernel (evaluate_dp_work_A) that will determine if new nd-range work needs to be performed. If new nd-range work does need to be performed, then evaluate_dp_work_A will enqueue a new instance of dp_func_A . This process is repeated until all the work is completed.

```
kernel void
dp_func_A(queue_t q, ...)
{
    ...

    // queue a single instance of evaluate_dp_work_A to
    // device queue q. queued kernel begins execution after
    // kernel dp_func_A finishes

    if (get_global_id(0) == 0)
    {
        enqueue_kernel(q,
                       CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                       ndrange_1d(1),
                       ^{evaluate_dp_work_A(q, ...);});
    }
}

kernel void
evaluate_dp_work_A(queue_t q,...)
{
    // check if more work needs to be performed
    bool more_work = check_new_work(...);
    if (more_work)
    {
        size_t global_work_size = compute_global_size(...);

        void (^dp_func_A_blk)(void) =
            ^{dp_func_A(q, ...});

        // get local WG-size for kernel dp_func_A
        size_t local_work_size =
            get_kernel_work_group_size(dp_func_A_blk);

        // build nd-range descriptor
        ndrange_t ndrange = ndrange_1D(global_work_size,
                                       local_work_size);

        // enqueue dp_func_A
        enqueue_kernel(q,
                       CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                       ndrange,
                       dp_func_A_blk);
    }
    ...
}
```

Determining when a child kernel begins execution

The `kernel_enqueue_flags_t`[64] argument to enqueue_kernel built-in functions can be used to specify when the child kernel begins execution. Supported values are described in the table below:

[64] Implementations are not required to honor this flag. Implementations may not schedule kernel launch earlier than the point specified by this flag, however.

Table 31. Kernel Enqueue Flags

`kernel_enqueue_flags_t` **enum**	**Description**
`CLK_ENQUEUE_FLAGS_NO_WAIT`	Indicates that the enqueued kernels do not need to wait for the parent kernel to finish execution before they begin execution.
`CLK_ENQUEUE_FLAGS_WAIT_KERNEL`	Indicates that all work-items of the parent kernel must finish executing and all immediate[65] side effects committed before the enqueued child kernel may begin execution.
`CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP`[66]	Indicates that the enqueued kernels wait only for the workgroup that enqueued the kernels to finish before they begin execution.

[65] Immediate meaning not side effects resulting from child kernels. The side effects would include stores to `global` memory and pipe reads and writes.

[66] This acts as a memory synchronization point between work-items in a work-group and child kernels enqueued by work-items in the work-group.

> The `kernel_enqueue_flags_t` flags are useful when a kernel enqueued from the host and executing on a device enqueues kernels on the device. The kernel enqueued from the host may not have an event associated with it. The `kernel_enqueue_flags_t` flags allow the developer to indicate when the child kernels can begin execution.

Determining when a parent kernel has finished execution

A parent kernel's execution status is considered to be complete when it and all its child kernels have finished execution. The execution status of a parent kernel will be `CL_COMPLETE` if this kernel and all its child kernels finish execution successfully. The execution status of the kernel will be an error code (given by a negative integer value) if it or any of its child kernels encounter an error, or are abnormally terminated.

For example, assume that the host enqueues a kernel `k` for execution on a device. Kernel `k` when executing on the device enqueues kernels `A` and `B` to a device queue(s). The enqueue_kernel call to enqueue kernel `B` specifies the event associated with kernel `A` in the `event_wait_list` argument, i.e. wait for kernel `A` to finish execution before kernel `B` can begin execution. Let's assume kernel `A` enqueues kernels `X`, `Y` and `Z`. Kernel `A` is considered to have finished execution, i.e. its execution status is `CL_COMPLETE`, only after `A` and the kernels `A` enqueued (and any kernels these enqueued kernels enqueue and so on) have finished execution.

Built-in Functions - Kernel Query Functions

Table 32. Built-in Kernel Query Functions

Built-in Function	Description
uint **get_kernel_work_group_size**(void (^block)(void)) uint **get_kernel_work_group_size**(void (^block)(local void *, ...))	This provides a mechanism to query the maximum work-group size that can be used to execute a block on a specific device given by *device*. *block* specifies the block to be enqueued.
uint **get_kernel_preferred_ work_group_size_multiple**(void (^block)(void)) uint **get_kernel_preferred_ work_group_size_multiple**(void (^block)(local void *, ...))	Returns the preferred multiple of work-group size for launch. This is a performance hint. Specifying a work-group size that is not a multiple of the value returned by this query as the value of the local work size argument to enqueue_kernel will not fail to enqueue the block for execution unless the work-group size specified is larger than the device maximum.

Built-in Functions - Queuing other commands

The following table describes the list of built-in functions that can be used to enqueue commands such as a marker.

Table 33. Built-in Other Enqueue Functions

Built-in Function	Description
int **enqueue_marker**(queue_t *queue*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*)	Enqueue a marker command to *queue*. The marker command waits for a list of events specified by *event_wait_list* to complete before the marker completes. *event_ret* must not be `NULL` as otherwise this is a no-op. If an event is returned, **enqueue_marker** performs an implicit retain on the returned event.

The **enqueue_marker** built-in function returns `CLK_SUCCESS` if the marked command is enqueued successfully and returns `CLK_ENQUEUE_FAILURE` otherwise. If the -g compile option is specified in compiler options passed to **clCompileProgram** or **clBuildProgram**, the following errors may be returned instead of `CLK_ENQUEUE_FAILURE` to indicate why **enqueue_marker** failed to enqueue the marker command:

- `CLK_INVALID_QUEUE` if *queue* is not a valid device queue.
- `CLK_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL`, or if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CLK_DEVICE_QUEUE_FULL` if *queue* is full.

- `CLK_EVENT_ALLOCATION_FAILURE` if *event_ret* is not `NULL` and an event could not be allocated.
- `CLK_OUT_OF_RESOURCES` if there is a failure to queue the block in *queue* because of insufficient resources needed to execute the kernel.

Built-in Functions - Event Functions

The following table describes the list of built-in functions that work on events.

Table 34. Built-in Event Functions

Built-in Function	Description
void **retain_event**(clk_event_t *event*)	Increments the event reference count. *event* must be an event returned by enqueue_kernel or enqueue_marker or a user event.
void **release_event**(clk_event_t *event*)	Decrements the event reference count. The event object is deleted once the event reference count is zero, the specific command identified by this event has completed (or terminated) and there are no commands in any device command queue that require a wait for this event to complete. *event* must be an event returned by enqueue_kernel, enqueue_marker or a user event.
clk_event_t **create_user_event**()	Create a user event. Returns the user event. The execution status of the user event created is set to `CL_SUBMITTED`.
bool **is_valid_event**(clk_event_t *event*)	Returns *true* if *event* is a valid event. Otherwise returns *false*.
void **set_user_event_status**(clk_event_t *event*, int *status*)	Sets the execution status of a user event. *event* must be a user-event. *status* can be either `CL_COMPLETE` or a negative integer value indicating an error.

Built-in Function	Description
void **capture_event_profiling_info**(clk_event_t *event*, clk_profiling_info *name*, global void * *value*)	Captures the profiling information for functions that are enqueued as commands. The specific function being referred to is: enqueue_kernel. These enqueued commands are identified by unique event objects. The profiling information will be available in *value* once the command identified by *event* has completed. — *event* must be an event returned by enqueue_kernel. *name* identifies which profiling information is to be queried and can be: `CLK_PROFILING_COMMAND_EXEC_TIME` *value* is a pointer to two 64-bit values. The first 64-bit value describes the elapsed time `CL_PROFILING_COMMAND_END` - `CL_PROFLING_COMMAND_START` for the command identified by *event* in nanoseconds. The second 64-bit value describes the elapsed time `CL_PROFILING_COMMAND_COMPLETE` - `CL_PROFILING_COMAMND_START` for the command identified by *event* in nanoseconds. [NOTE] ==== The behavior of capture_event_profiling_info when called multiple times for the same *event* is undefined. ==== —

Events can be used to identify commands enqueued to a command-queue from the host. These events created by the OpenCL runtime can only be used on the host, i.e. as events passed in the *event_wait_list* argument to various **clEnqueue** APIs or runtime APIs that take events as arguments, such as **clRetainEvent**, **clReleaseEvent**, and **clGetEventProfilingInfo**.

Similarly, events can be used to identify commands enqueued to a device queue (from a kernel). These event objects cannot be passed to the host or used by OpenCL runtime APIs such as the **clEnqueue** APIs or runtime APIs that take event arguments.

clRetainEvent and **clReleaseEvent** will return `CL_INVALID_OPERATION` if *event* specified is an event that refers to any kernel enqueued to a device queue using **enqueue_kernel** or **enqueue_marker**, or is a user event created by **create_user_event**.

Similarly, **clSetUserEventStatus** can only be used to set the execution status of events created using **clCreateUserEvent**. User events created on the device can be set using set_user_event_status built-in function.

The example below shows how events can be used with kernels enqueued to multiple device queues.

```
extern void barA_kernel(...);
extern void barB_kernel(...);

kernel void
foo(queue_t q0, queue q1, ...)
{
    ...
    clk_event_t evt0;

    // enqueue kernel to queue q0
    enqueue_kernel(q0,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange_A,
                   0, NULL, &evt0,
                   ^{barA_kernel(...);} );

    // enqueue kernel to queue q1
    enqueue_kernel(q1,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange_B,
                   1, &evt0, NULL,
                   ^{barB_kernel(...);} );

    // release event evt0. This will get released
    // after barA_kernel enqueued in queue q0 has finished
    // execution and barB_kernel enqueued in queue q1 and
    // waits for evt0 is submitted for execution, i.e. wait
    // for evt0 is satisfied.
    release_event(evt0);

}
```

The example below shows how the marker command can be used with kernels enqueued to a device queue.

```
kernel void
foo(queue_t q, ...)
{
    ...
    clk_event_t marker_event;
    clk_event_t events[2];

    enqueue_kernel(q,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange,
                   0, NULL, &events[0],
                   ^{barA_kernel(...);} );

    enqueue_kernel(q,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange,
                   0, NULL, &events[1],
                   ^{barB_kernel(...);} );

    // barA_kernel and barB_kernel can be executed
    // out of order. we need to wait for both these
    // kernels to finish execution before barC_kernel
    // starts execution so we enqueue a marker command and
    // then enqueue barC_kernel that waits on the event
    // associated with the marker.
    enqueue_marker(q, 2, events, &marker_event);

    enqueue_kernel(q,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   1, &marker_event, NULL,
                   ^{barC_kernel(...);} );

    release_event(events[0];
    release_event(events[1]);
    release_event(marker_event);
}
```

Built-in Functions - Helper Functions

Table 35. Built-in Helper Functions

Built-in Function	Description
queue_t **get_default_queue**(void)	Returns the default device queue. If a default device queue has not been created, `CLK_NULL_QUEUE` is returned.

ndrange_t **ndrange_1D**(size_t *global_work_size*) ndrange_t **ndrange_1D**(size_t *global_work_size*, size_t *local_work_size*) ndrange_t **ndrange_1D**(size_t *global_work_offset*, size_t *global_work_size*, size_t *local_work_size*) ndrange_t **ndrange_2D**(const size_t *global_work_size*[2]) ndrange_t **ndrange_2D**(const size_t *global_work_size*[2], const size_t *local_work_size*[2]) ndrange_t **ndrange_2D**(const size_t *global_work_offset*[2], const size_t *global_work_size*[2], const size_t *local_work_size*[2]) ndrange_t **ndrange_3D**(const size_t *global_work_size*[3]) ndrange_t **ndrange_3D**(const size_t *global_work_size*[3], const size_t *local_work_size*[3]) ndrange_t **ndrange_3D**(const size_t *global_work_offset*[3], const size_t *global_work_size*[3], const size_t *local_work_size*[3])	Builds a 1D, 2D or 3D ND-range descriptor.

Chapter 7. OpenCL Numerical Compliance

This section describes features of the C99 and IEEE 754 standards that must be supported by all OpenCL compliant devices.

This section describes the functionality that must be supported by all OpenCL devices for single precision floating-point numbers. Currently, only single precision floating-point is a requirement. Double precision floating-point is an optional feature.

7.1. Rounding Modes

Floating-point calculations may be carried out internally with extra precision and then rounded to fit into the destination type. IEEE 754 defines four possible rounding modes:

- Round to nearest even
- Round toward +∞
- Round toward -∞
- Round toward zero

Round to nearest even is currently the only rounding mode required[67] by the OpenCL specification for single precision and double precision operations and is therefore the default rounding mode. In addition, only static selection of rounding mode is supported. Dynamically reconfiguring the rounding modes as specified by the IEEE 754 spec is unsupported.

[67] Except for the embedded profile whether either round to zero or round to nearest rounding mode may be supported for single precision floating-point.

7.2. INF, NaN and Denormalized Numbers

`INF` and NaNs must be supported. Support for signaling NaNs is not required.

Support for denormalized numbers with single precision floating-point is optional. Denormalized single precision floating-point numbers passed as input or produced as the output of single precision floating-point operations such as add, sub, mul, divide, and the functions defined in math functions, common functions, and geometric functions may be flushed to zero.

7.3. Floating-Point Exceptions

Floating-point exceptions are disabled in OpenCL. The result of a floating-point exception must match the IEEE 754 spec for the exceptions not enabled case. Whether and when the implementation sets floating-point flags or raises floating-point exceptions is implementation-defined. This standard provides no method for querying, clearing or setting floating-point flags or trapping raised exceptions. Due to non-performance, non-portability of trap mechanisms and the impracticality of servicing precise exceptions in a vector context (especially on heterogeneous hardware), such features are discouraged.

Implementations that nevertheless support such operations through an extension to the standard shall initialize with all exception flags cleared and the exception masks set so that exceptions raised by arithmetic operations do not trigger a trap to be taken. If the underlying work is reused by the implementation, the implementation is however not responsible for reclearing the flags or resetting exception masks to default values before entering the kernel. That is to say that kernels that do not inspect flags or enable traps are licensed to expect that their arithmetic will not trigger a trap. Those kernels that do examine flags or enable traps are responsible for clearing flag state and disabling all traps before returning control to the implementation. Whether or when the underlying work-item (and accompanying global floating-point state if any) is reused is implementation-defined.

The expressions **math_errorhandling** and `MATH_ERREXCEPT` are reserved for use by this standard, but not defined. Implementations that extend this specification with support for floating-point exceptions shall define **math_errorhandling** and `MATH_ERREXCEPT` per TC2 to the C99 Specification.

7.4. Relative Error as ULPs

In this section we discuss the maximum relative error defined as ulp (units in the last place). Addition, subtraction, multiplication, fused multiply-add and conversion between integer and a single precision floating-point format are IEEE 754 compliant and are therefore correctly rounded. Conversion between floating-point formats and explicit conversions must be correctly rounded.

The ULP is defined as follows:

> If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then ulp(x) = $|b - a|$, otherwise ulp(x) is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, ulp(NaN) is NaN.

Attribution: This definition was taken with consent from Jean-Michel Muller with slight clarification for behavior at zero. Refer to _ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf._

The following table[68] describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

[68] The ULP values for built-in math functions **lgamma** and **lgamma_r** is currently undefined.

Table 36. ULP values for single precision built-in math functions

Function	**Min Accuracy - ULP values**[69]
$x + y$	Correctly rounded
$x - y$	Correctly rounded
$x * y$	Correctly rounded
1.0 / x	≤ 2.5 ulp
x / y	≤ 2.5 ulp

acos	≤ 4 ulp
acospi	≤ 5 ulp
asin	≤ 4 ulp
asinpi	≤ 5 ulp
atan	≤ 5 ulp
atan2	≤ 6 ulp
atanpi	≤ 5 ulp
atan2pi	≤ 6 ulp
acosh	≤ 4 ulp
asinh	≤ 4 ulp
atanh	≤ 5 ulp
cbrt	≤ 2 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	≤ 4 ulp
cosh	≤ 4 ulp
cospi	≤ 4 ulp
erfc	≤ 16 ulp
erf	≤ 16 ulp
exp	≤ 3 ulp
exp2	≤ 3 ulp
exp10	≤ 3 ulp
expm1	≤ 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	≤ 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	≤ 3 ulp
log2	≤ 3 ulp

log10	≤ 3 ulp
log1p	≤ 2 ulp
logb	0 ulp
mad	Implemented either as a correctly rounded fma or as a multiply followed by an add both of which are correctly rounded
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	≤ 16 ulp
pown(x, y)	≤ 16 ulp
powr(x, y)	≤ 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	≤ 16 ulp
round	Correctly rounded
rsqrt	≤ 2 ulp
sin	≤ 4 ulp
sincos	≤ 4 ulp for sine and cosine values
sinh	≤ 4 ulp
sinpi	≤ 4 ulp
sqrt	≤ 3 ulp
tan	≤ 5 ulp
tanh	≤ 5 ulp
tanpi	≤ 6 ulp
tgamma	≤ 16 ulp
trunc	Correctly rounded
half_cos	≤ 8192 ulp
half_divide	≤ 8192 ulp
half_exp	≤ 8192 ulp
half_exp2	≤ 8192 ulp
half_exp10	≤ 8192 ulp
half_log	≤ 8192 ulp
half_log2	≤ 8192 ulp

half_log10	≤ 8192 ulp
half_powr	≤ 8192 ulp
half_recip	≤ 8192 ulp
half_rsqrt	≤ 8192 ulp
half_sin	≤ 8192 ulp
half_sqrt	≤ 8192 ulp
half_tan	≤ 8192 ulp
native_cos	Implementation-defined
native_divide	Implementation-defined
native_exp	Implementation-defined
native_exp2	Implementation-defined
native_exp10	Implementation-defined
native_log	Implementation-defined
native_log2	Implementation-defined
native_log10	Implementation-defined
native_powr	Implementation-defined
native_recip	Implementation-defined
native_rsqrt	Implementation-defined
native_sin	Implementation-defined
native_sqrt	Implementation-defined
native_tan	Implementation-defined

[69] 0 ulp is used for math functions that do not require rounding.

The following table describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values for the embedded profile. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Table 37. ULP values for the embedded profile

Function	**Min Accuracy - ULP values**[70]
x + *y*	Correctly rounded
x - *y*	Correctly rounded
x * *y*	Correctly rounded
1.0 / *x*	≤ 3 ulp
x / *y*	≤ 3 ulp
acos	≤ 4 ulp
acospi	≤ 5 ulp
asin	≤ 4 ulp

asinpi	≤ 5 ulp
atan	≤ 5 ulp
atan2	≤ 6 ulp
atanpi	≤ 5 ulp
atan2pi	≤ 6 ulp
acosh	≤ 4 ulp
asinh	≤ 4 ulp
atanh	≤ 5 ulp
cbrt	≤ 4 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	≤ 4 ulp
cosh	≤ 4 ulp
cospi	≤ 4 ulp
erfc	≤ 16 ulp
erf	≤ 16 ulp
exp	≤ 4 ulp
exp2	≤ 4 ulp
exp10	≤ 4 ulp
expm1	≤ 4 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	≤ 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	≤ 4 ulp
log2	≤ 4 ulp
log10	≤ 4 ulp
log1p	≤ 4 ulp
logb	0 ulp

mad	Any value allowed (infinite ulp)
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	≤ 16 ulp
pown(x, y)	≤ 16 ulp
powr(x, y)	≤ 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	≤ 16 ulp
round	Correctly rounded
rsqrt	≤ 4 ulp
sin	≤ 4 ulp
sincos	≤ 4 ulp for sine and cosine values
sinh	≤ 4 ulp
sinpi	≤ 4 ulp
sqrt	≤ 4 ulp
tan	≤ 5 ulp
tanh	≤ 5 ulp
tanpi	≤ 6 ulp
tgamma	≤ 16 ulp
trunc	Correctly rounded
half_cos	≤ 8192 ulp
half_divide	≤ 8192 ulp
half_exp	≤ 8192 ulp
half_exp2	≤ 8192 ulp
half_exp10	≤ 8192 ulp
half_log	≤ 8192 ulp
half_log2	≤ 8192 ulp
half_log10	≤ 8192 ulp
half_powr	≤ 8192 ulp
half_recip	≤ 8192 ulp
half_rsqrt	≤ 8192 ulp
half_sin	≤ 8192 ulp

half_sqrt	≤ 8192 ulp
half_tan	≤ 8192 ulp
native_cos	Implementation-defined
native_divide	Implementation-defined
native_exp	Implementation-defined
native_exp2	Implementation-defined
native_exp10	Implementation-defined
native_log	Implementation-defined
native_log2	Implementation-defined
native_log10	Implementation-defined
native_powr	Implementation-defined
native_recip	Implementation-defined
native_rsqrt	Implementation-defined
native_sin	Implementation-defined
native_sqrt	Implementation-defined
native_tan	Implementation-defined

[70] 0 ulp is used for math functions that do not require rounding.

The following table describes the minimum accuracy of commonly used single precision floating-point arithmetic operations given as ULP values if the `-cl-unsafe-math-optimizations` compiler option is specified when compiling or building an OpenCL program. For derived implementations, the operations used in the derivation may themselves be relaxed according to the following table. The minimum accuracy of math functions not defined in the following table when the `-cl-unsafe-math-optimizations` compiler option is specified is as defined in ULP values for single precision built-in math functions when operating in the full profile, and as defined in ULP values for the embedded profile when operating in the embedded profile. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Table 38. ULP values for single precision built-in math functions with unsafe math optimizations in the full and embedded profiles

Function	Min Accuracy - ULP values[71]
1.0 / ***x***	≤ 2.5 ulp for *x* in the domain of 2^{-126} to 2^{126} for the full profile, and ≤ 3 ulp for the embedded profile.
x / *y*	≤ 2.5 ulp for *x* in the domain of 2^{-62} to 2^{62} and *y* in the domain of 2^{-62} to 2^{62} for the full profile, and ≤ 3 ulp for the embedded profile.
acos(*x*)	≤ 4096 ulp
acospi(*x*)	Implemented as **acos**(*x*) * `M_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
asin(*x*)	≤ 4096 ulp

asinpi(*x*)	Implemented as **asin**(*x*) * `M_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
atan(*x*)	≤ 4096 ulp
atan2(*y*, *x*)	Implemented as **atan**(*y* / *x*) for *x* > 0, **atan**(*y* / *x*) `M_1_PI_F` for *x* < 0 and *y* > 0 and **atan**(*y* / *x*) - `M_1_PI_F` for *x* < 0 and *y* < 0.
atanpi(*x*)	Implemented as **atan**(*x*) * `M_1_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
atan2pi(*y*, *x*)	Implemented as **atan2**(*y*, *x*) * `M_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
acosh(*x*)	Implemented as **log**(*x* + **sqrt**(*x* * *x* - 1)).
asinh(*x*)	Implemented as **log**(*x* + **sqrt**(*x* * *x* + 1)).
cbrt(*x*)	Implemented as **rootn**(*x*, 3). For non-derived implementations, the error is ≤ 8192 ulp.
cos(*x*)	For *x* in the domain [-π, π], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
cosh(*x*)	Defined for *x* in the domain [-88,88] and implemented as 0.5f * (**exp**(*x*) + **exp**(-*x*)). For non-derived implementations, the error is ≤ 8192 ULP.
cospi(*x*)	For *x* in the domain [-1, 1], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
exp(*x*)	≤ 3 + **floor**(**fabs**(2 * *x*)) ulp for the full profile, and ≤ 4 ulp for the embedded profile.
exp2(*x*)	≤ 3 + **floor**(**fabs**(2 * *x*)) ulp for the full profile, and ≤ 4 ulp for the embedded profile.
exp10(*x*)	Derived implementations implement this as **exp2**(*x* * **log2**(10)). For non-derived implementations, the error is ≤ 8192 ulp.
expm1(*x*)	Derived implementations implement this as **exp**(*x*) - 1. For non-derived implementations, the error is ≤ 8192 ulp.
log(*x*)	For *x* in the domain [0.5, 2] the maximum absolute error is ≤ 2^{-21}; otherwise the maximum error is ≤3 ulp for the full profile and ≤ 4 ulp for the embedded profile
log2(*x*)	For *x* in the domain [0.5, 2] the maximum absolute error is ≤ 2^{-21}; otherwise the maximum error is ≤3 ulp for the full profile and ≤ 4 ulp for the embedded profile
log10(*x*)	For *x* in the domain [0.5, 2] the maximum absolute error is ≤ 2^{-21}; otherwise the maximum error is ≤3 ulp for the full profile and ≤ 4 ulp for the embedded profile

log1p(*x*)	Derived implementations implement this as **log** (*x* + 1). For non-derived implementations, the error is ≤ 8192 ulp.
pow(*x*, *y*)	Undefined for *x* = 0 and *y* = 0. Undefined for *x* < 0 and non-integer y. Undefined for *x* < 0 and *y* outside the domain [-2^24, 2^24]. For *x* > 0 or *x* < 0 and even *y*, derived implementations implement this as **exp2**(y * log2(*x*)). For *x* < 0 and odd *y*, derived implementations implement this as **-exp2(y * *log2**(fabs(*x*))[72]. For *x* == 0 and nonzero *y*, derived implementations return zero. For non-derived implementations, the error is ≤ 8192 ULP
pown(*x*, *y*)	Defined only for integer values of y. Undefined for *x* = 0 and *y* = 0. For *x* >= 0 or *x* < 0 and even *y*, derived implementations implement this as **exp2**(*y* * **log2**(*x*)). For *x* < 0 and odd *y*, derived implementations implement this as **-exp2**(*y* * **log2**(**fabs**(*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
powr(*x*, *y*)	Defined only for *x* >= 0. Undefined for *x* = 0 and *y* = 0. Derived implementations implement this as **exp2**(*y* * **log2**(*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
rootn(*x*, *y*)	Defined for *x* > 0 when *y* is nonzero, derived implementations implement this case as **exp2**(log2(*x*) / *y*). Defined for *x* < 0 when *y* is odd, derived implementations implement this case as **-exp2**(**log2**(-*x*) / *y*). Defined for *x* = +/-0 when *y* > 0, derived implementations will return +0 in this case. For non-derived implementations, the error is ≤ 8192 ULP.
sin(*x*)	For *x* in the domain [-π, π], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
sincos(*x*)	ulp values as defined for **sin**(*x*) and **cos**(*x*)
sinh(*x*)	Defined for *x* in the domain [-88,88]. For *x* in [-2 $^{-10,2}$-10], derived implementations implement as *x*. For *x* outside of [-2^{10,2}10], derived implement as **0.5f * (*exp**(*x*) - **exp**(-*x*)). For non-derived implementations, the error is ≤ 8192 ULP.
sinpi(*x*)	For *x* in the domain [-1, 1], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
tan(*x*)	Derived implementations implement this as **sin** (*x*) * (`1.0f` / **cos**(*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
tanpi(*x*)	Derived implementations implement this as **tan** (*x* * `M_PI_F`). For non-derived implementations, the error is ≤ 8192 ulp for *x* in the domain [-1, 1].

x * *y* + *z*	Implemented either as a correctly rounded **fma** or as a multiply and an add both of which are correctly rounded.

[71] 0 ulp is used for math functions that do not require rounding.

[72] On some implementations, **powr**() or **pown**() may perform faster than **pow**(). If *x* is known to be >= 0, consider using **powr**() in place of **pow**(), or if *y* is known to be an integer, consider using **pown**() in place of **pow**().

The following table describes the minimum accuracy of double precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result.

Table 39. ULP values for double precision built-in math functions

Function	Min Accuracy - ULP values[73]
x + *y*	Correctly rounded
x - *y*	Correctly rounded
x * *y*	Correctly rounded
1.0 / *x*	Correctly rounded
x / *y*	Correctly rounded
acos	≤ 4 ulp
acospi	≤ 5 ulp
asin	≤ 4 ulp
asinpi	≤ 5 ulp
atan	≤ 5 ulp
atan2	≤ 6 ulp
atanpi	≤ 5 ulp
atan2pi	≤ 6 ulp
acosh	≤ 4 ulp
asinh	≤ 4 ulp
atanh	≤ 5 ulp
cbrt	≤ 2 ulp
ceil	Correctly rounded
copysign	0 ulp
cos	≤ 4 ulp
cosh	≤ 4 ulp
cospi	≤ 4 ulp
erfc	≤ 16 ulp
erf	≤ 16 ulp

exp	≤ 3 ulp
exp2	≤ 3 ulp
exp10	≤ 3 ulp
expm1	≤ 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	≤ 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
log	≤ 3 ulp
log2	≤ 3 ulp
log10	≤ 3 ulp
log1p	≤ 2 ulp
logb	0 ulp
mad	Any value allowed (infinite ulp)
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	≤ 16 ulp
pown(x, y)	≤ 16 ulp
powr(x, y)	≤ 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	≤ 16 ulp
round	Correctly rounded
rsqrt	≤ 2 ulp
sin	≤ 4 ulp

sincos	≤ 4 ulp for sine and cosine values
sinh	≤ 4 ulp
sinpi	≤ 4 ulp
fsqrt	Correctly rounded
tan	≤ 5 ulp
tanh	≤ 5 ulp
tanpi	≤ 6 ulp
tgamma	≤ 16 ulp
trunc	Correctly rounded

[73] 0 ulp is used for math functions that do not require rounding.

7.5. Edge Case Behavior

The edge case behavior of the math functions shall conform to sections F.9 and G.6 of the C99 Specification, except where noted below.

7.5.1. Additional Requirements Beyond C99 TC2

Functions that return a NaN with more than one NaN operand shall return one of the NaN operands. Functions that return a NaN operand may silence the NaN if it is a signaling NaN. A non-signaling NaN shall be converted to a non-signaling NaN. A signaling NaN shall be converted to a NaN, and should be converted to a non-signaling NaN. How the rest of the NaN payload bits or the sign of NaN is converted is undefined.

half_<funcname> functions behave identically to the function of the same name without the **half_** prefix. They must conform to the same edge case requirements (see sections F.9 and G.6 of the C99 Specification). For other cases, except where otherwise noted, these single precision functions are permitted to have up to 8192 ulps of error (as measured in the single precision result), although better accuracy is encouraged.

The usual allowances for rounding error or flushing behavior shall not apply for those values for which section F.9 of the C99 Specification, or the additional requirements and edge case behavior below (and similar sections for other floating-point precisions) prescribe a result (e.g. **ceil**(-1 < *x* < 0) returns -0). Those values shall produce exactly the prescribed answers, and no other. Where the ± symbol is used, the sign shall be preserved. For example, **sin**(±0) = ±0 shall be interpreted to mean **sin**(+0) is +0 and **sin**(-0) is -0.

> **acospi**(1) = +0.
>
> **acospi**(*x*) returns a NaN for |*x*| > 1.
>
> **asinpi**(±0) = ±0.
>
> **asinpi**(*x*) returns a NaN for |*x*| > 1.

atanpi(±0) = ±0.

atanpi(±∞) = ±0.5.

atan2pi(±0, -0) = ±1.

atan2pi(±0, +0) = ±0.

atan2pi(±0, x) returns ±1 for $x < 0$.

atan2pi(±0, x) returns ±0 for $x > 0$.

atan2pi(y, ±0) returns -0.5 for $y < 0$.

atan2pi(y, ±0) returns 0.5 for $y > 0$.

atan2pi(±_y_, -∞) returns ±1 for finite $y > 0$.

atan2pi(±_y_, +∞) returns ±0 for finite $y > 0$.

atan2pi(±∞, x) returns ±0.5 for finite x.

atan2pi(±∞, -∞) returns ±0.75.

atan2pi(±∞, +∞) returns ±0.25.

ceil($-1 < x < 0$) returns -0.

cospi(±0) returns 1

cospi($n + 0.5$) is +0 for any integer n where $n + 0.5$ is representable.

cospi(±∞) returns a NaN.

exp10(-∞) returns +0.

exp10(+∞) returns +∞.

distance(x, y) calculates the distance from x to y without overflow or extraordinary precision loss due to underflow.

fdim(any, NaN) returns NaN.

fdim(NaN, any) returns NaN.

fmod(±0, NaN) returns NaN.

frexp(±∞, *exp*) returns ±∞ and stores 0 in *exp*.

frexp(NaN, *exp*) returns the NaN and stores 0 in *exp*.

fract(*x*, *iptr*) shall not return a value greater than or equal to 1.0, and shall not return a value less than 0.

fract(+0, *iptr*) returns +0 and +0 in iptr.

fract(-0, *iptr*) returns -0 and -0 in iptr.

fract(+∞, *iptr*) returns +0 and +∞ in *iptr*.

fract(-∞, *iptr*) returns -0 and -∞ in *iptr*.

fract(NaN, *iptr*) returns the NaN and NaN in *iptr*.

length calculates the length of a vector without overflow or extraordinary precision loss due to underflow.

lgamma_r(*x*, *signp*) returns 0 in *signp* if *x* is zero or a negative integer.

nextafter(-0, *y* > 0) returns smallest positive denormal value.

nextafter(+0, *y* < 0) returns smallest negative denormal value.

normalize shall reduce the vector to unit length, pointing in the same direction without overflow or extraordinary precision loss due to underflow.

normalize(*v*) returns *v* if all elements of *v* are zero.

normalize(*v*) returns a vector full of NaNs if any element is a NaN.

normalize(*v*) for which any element in *v* is infinite shall proceed as if the elements in *v* were replaced as follows:

```
for (i = 0; i < sizeof(v) / sizeof(v[0]); i++)
   v[i] = isinf(v[i]) ? copysign(1.0, v[i]) : 0.0 * v[i];
```

pow(±0, -∞) returns +∞

pown(*x*, 0) is 1 for any *x*, even zero, NaN or infinity.

pown(±0, *n*) is ±∞ for odd *n* < 0.

pown(±0, n) is +∞ for even $n < 0$.

pown(±0, n) is +0 for even $n > 0$.

pown(±0, n) is ±0 for odd $n > 0$.

powr(x, ±0) is 1 for finite $x > 0$.

powr(±0, y) is +∞ for finite $y < 0$.

powr(±0, -∞) is +∞.

powr(±0, y) is +0 for $y > 0$.

powr(+1, y) is 1 for finite y.

powr(x, y) returns NaN for $x < 0$.

powr(±0, ±0) returns NaN.

powr(+∞, ±0) returns NaN.

powr(+1, ±∞) returns NaN.

powr(x, NaN) returns the NaN for $x >= 0$.

powr(NaN, y) returns the NaN.

rint(-0.5 <= x < 0) returns -0.

remquo(x, y, &_quo_) returns a NaN and 0 in *quo* if x is ±∞, or if y is 0 and the other argument is non-NaN or if either argument is a NaN.

rootn(±0, n) is ±∞ for odd $n < 0$.

rootn(±0, n) is +∞ for even $n < 0$.

rootn(±0, n) is +0 for even $n > 0$.

rootn(±0, n) is ±0 for odd $n > 0$.

rootn(x, n) returns a NaN for $x < 0$ and n is even.

rootn(x, 0) returns a NaN.

round(-0.5 < x < 0) returns -0.

sinpi(±0) returns ±0.

sinpi(+n) returns +0 for positive integers n.

sinpi(-n) returns -0 for negative integers n.

sinpi(±∞) returns a NaN.

tanpi(±0) returns ±0.

tanpi(±∞) returns a NaN.

tanpi(n) is **copysign**(0.0, n) for even integers n.

tanpi(n) is **copysign**(0.0, - n) for odd integers n.

tanpi(n + 0.5) for even integer n is +∞ where n + 0.5 is representable.

tanpi(n + 0.5) for odd integer n is -∞ where n + 0.5 is representable.

trunc($-1 < x < 0$) returns -0. Binary file (standard input) matches

7.5.2. Changes to C99 TC2 Behavior

modf behaves as though implemented by:

```
gentype modf(gentype value, gentype *iptr)
{
    *iptr = trunc( value );
    return copysign(isinf( value ) ? 0.0 : value - *iptr, value);
}
```

rint always rounds according to round to nearest even rounding mode even if the caller is in some other rounding mode.

7.5.3. Edge Case Behavior in Flush To Zero Mode

If denormals are flushed to zero, then a function may return one of four results:

1. Any conforming result for non-flush-to-zero mode
2. If the result given by 1. is a sub-normal before rounding, it may be flushed to zero
3. Any non-flushed conforming result for the function if one or more of its sub-normal operands are flushed to zero.
4. If the result of 3. is a sub-normal before rounding, the result may be flushed to zero.

In each of the above cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

If subnormals are flushed to zero, a device may choose to conform to the following edge cases for **nextafter** instead of those listed in the additional requirements section.

> **nextafter**(+smallest normal, *y* < +smallest normal) = +0.
>
> **nextafter**(-smallest normal, *y* > -smallest normal) = -0.
>
> **nextafter**(-0, *y* > 0) returns smallest positive normal value.
>
> **nextafter**(+0, *y* < 0) returns smallest negative normal value.

For clarity, subnormals or denormals are defined to be the set of representable numbers in the range 0 < *x* < `TYPE_MIN` and `-TYPE_MIN` < *x* < -0. They do not include ±0. A non-zero number is said to be sub-normal before rounding if after normalization, its radix-2 exponent is less than (`TYPE_MIN_EXP` - 1)[74].

[74] Here `TYPE_MIN` and `TYPE_MIN_EXP` should be substituted by constants appropriate to the floating-point type under consideration, such as `FLT_MIN` and `FLT_MIN_EXP` for `float`.

Chapter 8. Image Addressing and Filtering

Let w_t, h_t and d_t be the width, height (or image array size for a 1D image array) and depth (or image array size for a 2D image array) of the image in pixels. Let *coord.xy* (also referred to as (*s*,*t*)) or *coord.xyz* (also referred to as (*s*,*t*,*r*)) be the coordinates specified to **read_image{f|i|ui}**. The sampler specified in **read_image{f|i|ui}** is used to determine how to sample the image and return an appropriate color.

8.1. Image Coordinates

This affects the interpretation of image coordinates. If image coordinates specified to **read_image{f|i|ui}** are normalized (as specified in the sampler), the *s*, *t*, and *r* coordinate values are multiplied by w_t, $h_{t,}$ and d_t respectively to generate the unnormalized coordinate values. For image arrays, the image array coordinate (i.e. *t* if it is a 1D image array or *r* if it is a 2D image array) specified to **read_image{f|i|ui}** must always be the un-normalized image coordinate value.

Let (*u*,*v*,*w*) represent the unnormalized image coordinate values.

8.2. Addressing and Filter Modes

We first describe how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is not `CLK_ADDRESS_REPEAT` nor `CLK_ADDRESS_MIRRORED_REPEAT`.

After generating the image coordinate (*u*,*v*,*w*) we apply the appropriate addressing and filter mode to generate the appropriate sample locations to read from the image.

If values in (*u*,*v*,*w*) are `INF` or NaN, the behavior of **read_image{f|i|ui}** is undefined.

Filter Mode `CLK_FILTER_NEAREST`

When filter mode is `CLK_FILTER_NEAREST`, the image element in the image that is nearest (in Manhattan distance) to that specified by (*u*,*v*,*w*) is obtained. This means the image element at location (*i*,*j*,*k*) becomes the image element value, where

```
i = address_mode((int)floor(u))
j = address_mode((int)floor(v))
k = address_mode((int)floor(w))
```

For a 3D image, the image element at location (*i*,*j*,*k*) becomes the color value. For a 2D image, the image element at location (*i*,*j*) becomes the color value.

The following table describes the address_mode function.

Table 40. Addressing modes to generate texel location

Addressing Mode	**Result of address_mode(coord)**
`CLK_ADDRESS_CLAMP_TO_EDGE`	clamp (coord, 0, size - 1)

`CLK_ADDRESS_CLAMP`	clamp (coord, -1, size)
`CLK_ADDRESS_NONE`	coord

The `size` term in this table is w_t for u, h_t for v and d_t for w.

The `clamp` function used in this table is defined as:

```
clamp(a, b, c) = return (a < b) ? b : ((a > c) ? c : a)
```

If the selected texel location (i,j,k) refers to a location outside the image, the border color is used as the color value for this texel.

Filter Mode `CLK_FILTER_LINEAR`

When filter mode is `CLK_FILTER_LINEAR`, a 2×2 square of image elements for a 2D image or a 2×2×2 cube of image elements for a 3D image is selected. This 2×2 square or 2×2×2 cube is obtained as follows.

Let

```
i0 = address_mode((int)floor(u - 0.5))
j0 = address_mode((int)floor(v - 0.5))
k0 = address_mode((int)floor(w - 0.5))
i1 = address_mode((int)floor(u - 0.5) + 1)
j1 = address_mode((int)floor(v - 0.5) + 1)
k1 = address_mode((int)floor(w - 0.5) + 1)
a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of x and is computed as `x - floor(x)`.

For a 3D image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

where `T_ijk` is the image element at location (i,j,k) in the 3D image.

For a 2D image, the image element value is found as

```
T = (1 - a) * (1 - b) * T_i0j0
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

where T_ij is the image element at location (i,j) in the 2D image.

If any of the selected T_ijk or T_ij in the above equations refers to a location outside the image, the border color is used as the color value for T_ijk or T_ij.

If the image channel type is CL_FLOAT or CL_HALF_FLOAT and any of the image elements T_ijk or T_ij is INF or NaN, the behavior of the built-in image read function is undefined.

We now discuss how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is CLK_ADDRESS_REPEAT.

If values in (s,t,r) are INF or NaN, the behavior of the built-in image read functions is undefined.

Filter Mode CLK_FILTER_NEAREST

When filter mode is CLK_FILTER_NEAREST, the image element at location (i,j,k) becomes the image element value, with i, j, and k computed as

```
u = (s - floor(s)) * w_t
i = (int)floor(u)
if (i > w_t - 1)
    i = i - w_t

v = (t - floor(t)) * h_t
j = (int)floor(v)
if (j > h_t - 1)
    j = j - h_t

w = (r - floor(r)) * d_t
k = (int)floor(w)
if (k > d_t - 1)
    k = k - d_t
```

For a 3D image, the image element at location (i,j,k) becomes the color value. For a 2D image, the image element at location (i,j) becomes the color value.

Filter Mode CLK_FILTER_LINEAR

When filter mode is CLK_FILTER_LINEAR, a 2×2 square of image elements for a 2D image or a 2×2×2 cube of image elements for a 3D image is selected. This 2×2 square or 2×2×2 cube is obtained as follows.

Let

```
u = (s - floor(s)) * w_t
i0 = (int)floor(u - 0.5)
i1 = i0 + 1
if (i0 < 0)
    i0 = w_t + i0
if (i1 > w_t - 1)
    i1 = i1 - w_t

v = (t - floor(t)) * h_t
j0 = (int)floor(v - 0.5)
j1 = j0 + 1
if (j0 < 0)
    j0 = h_t + j0
if (j1 > h_t - 1)
    j1 = j1 - h_t

w = (r - floor(r)) * d_t
k0 = (int)floor(w - 0.5)
k1 = k0 + 1
if (k0 < 0)
    k0 = d_t + k0
if (k1 > d_t - 1)
    k1 = k1 - d_t

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of x and is computed as `x - floor(x)`.

For a 3D image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

where `T_ijk` is the image element at location (*i*,*j*,*k*) in the 3D image.

For a 2D image, the image element value is found as

```
T = (1 - a) * (1 - b) * T_i0j0
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

where `T_ij` is the image element at location (*i*,*j*) in the 2D image.

If the image channel type is `CL_FLOAT` or `CL_HALF_FLOAT` and any of the image elements `T_ijk` or `T_ij` is `INF` or NaN, the behavior of the built-in image read function is undefined.

We now discuss how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is `CLK_ADDRESS_MIRRORED_REPEAT`. The `CLK_ADDRESS_MIRRORED_REPEAT` addressing mode causes the image to be read as if it is tiled at every integer seam with the interpretation of the image data flipped at each integer crossing. For example, the (*s*,*t*,*r*) coordinates between 2 and 3 are addressed into the image as coordinates from 1 down to 0. If values in (*s*,*t*,*r*) are `INF` or NaN, the behavior of the built-in image read functions is undefined.

Filter Mode `CLK_FILTER_NEAREST`

When filter mode is `CLK_FILTER_NEAREST`, the image element at location (*i*,*j*,*k*) becomes the image element value, with *i*,*j* and k computed as

```
s' = 2.0f * rint(0.5f * s)
s' = fabs(s - s')
u = s' * w_t
i = (int)floor(u)
i = min(i, w_t - 1)

t' = 2.0f * rint(0.5f * t)
t' = fabs(t - t')
v = t' * h_t
j = (int)floor(v)
j = min(j, h_t - 1)

r' = 2.0f * rint(0.5f * r)
r' = fabs(r - r')
w = r' * d_t
k = (int)floor(w)
k = min(k, d_t - 1)
```

For a 3D image, the image element at location (*i*,*j*,*k*) becomes the color value. For a 2D image, the image element at location (*i*,*j*) becomes the color value.

Filter Mode `CLK_FILTER_LINEAR`

When filter mode is `CLK_FILTER_LINEAR`, a 2×2 square of image elements for a 2D image or a 2×2×2 cube of image elements for a 3D image is selected. This 2×2 square or 2×2×2 cube is obtained as follows.

Let

```
s' = 2.0f * rint(0.5f * s)
s' = fabs(s - s')
u = s' * w_t
i0 = (int)floor(u - 0.5f)
i1 = i0 + 1
i0 = max(i0, 0)
i1 = min(i1, w_t - 1)

t' = 2.0f * rint(0.5f * t)
t' = fabs(t - t')
v = t' * h_t
j0 = (int)floor(v - 0.5f)
j1 = j0 + 1
j0 = max(j0, 0)
j1 = min(j1, h_t - 1)

r' = 2.0f * rint(0.5f * r)
r' = fabs(r - r')
w = r' * d_t
k0 = (int)floor(w - 0.5f)
k1 = k0 + 1
k0 = max(k0, 0)
k1 = min(k1, d_t - 1)

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of x and is computed as `x - floor(x)`.

For a 3D image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

where `T_ijk` is the image element at location (*i*,*j*,*k*) in the 3D image.

For a 2D image, the image element value is found as

```
T = (1 - a) * (1 - b) * T_i0j0
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

where `T_ij` is the image element at location (*i*,*j*) in the 2D image.

For a 1D image, the image element value is found as

```
T = (1 - a) * T_i0
    + a * T_i1
```

where `T_i` is the image element at location (*i*) in the 1D image.

If the image channel type is `CL_FLOAT` or `CL_HALF_FLOAT` and any of the image elements `T_ijk` or `T_ij` is `INF` or NaN, the behavior of the built-in image read function is undefined.

> If the sampler is specified as using unnormalized coordinates (floating-point or integer coordinates), filter mode set to `CLK_FILTER_NEAREST` and addressing mode set to one of the following modes - `CLK_ADDRESS_NONE`, `CLK_ADDRESS_CLAMP_TO_EDGE` or `CLK_ADDRESS_CLAMP`, the location of the image element in the image given by (*i*,*j*,*k*) will be computed without any loss of precision.
>
> For all other sampler combinations of normalized or unnormalized coordinates, filter and addressing modes, the relative error or precision of the addressing mode calculations and the image filter operation are not defined by this revision of the OpenCL specification. To ensure a minimum precision of image addressing and filter calculations across any OpenCL device, for these sampler combinations, developers should unnormalize the image coordinate in the kernel and implement the linear filter in the kernel with appropriate calls to **read_image{f|i|ui}** with a sampler that uses unnormalized coordinates, filter mode set to `CLK_FILTER_NEAREST`, addressing mode set to `CLK_ADDRESS_NONE`, `CLK_ADDRESS_CLAMP_TO_EDGE` *or* `CLK_ADDRESS_CLAMP`, and finally performing the interpolation of color values read from the image to generate the filtered color value.

8.3. Conversion Rules

In this section we discuss conversion rules that are applied when reading and writing images in a kernel.

8.3.1. Conversion rules for normalized integer channel data types

In this section we discuss converting normalized integer channel data types to floating-point values and vice-versa.

Converting normalized integer channel data types to floating-point values

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, **read_imagef** will convert the channel values from an 8-bit or 16-bit unsigned integer to normalized floating-point values in the range [`0.0f`, `1.0`].

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, **read_imagef** will convert the channel values from an 8-bit or 16-bit signed integer to normalized floating-point values in the range [`-1.0`, `1.0`].

These conversions are performed as follows:

`CL_UNORM_INT8` (8-bit unsigned integer) → `float`

normalized `float` value = `(float)c / 255.0f`

`CL_UNORM_INT_101010` (10-bit unsigned integer) → `float`

normalized `float` value = `(float)c / 1023.0f`

`CL_UNORM_INT16` (16-bit unsigned integer) → `float`

normalized `float` value = `(float)c / 65535.0f`

`CL_SNORM_INT8` (8-bit signed integer) → `float`

normalized `float` value = **max**(`-1.0f`, `(float)c / 127.0f`)

`CL_SNORM_INT16` (16-bit signed integer) → `float`

normalized `float` value = **max**(`-1.0f`, `(float)c / 32767.0f`)

The precision of the above conversions is <= 1.5 ulp except for the following cases.

For `CL_UNORM_INT8`

0 must convert to `0.0f` and

255 must convert to `1.0f`

For `CL_UNORM_INT_101010`

0 must convert to `0.0f` and

1023 must convert to `1.0f`

For `CL_UNORM_INT16`

0 must convert to `0.0f` and

65535 must convert to `1.0f`

For `CL_SNORM_INT8`

-128 and -127 must convert to `-1.0f`,

0 must convert to `0.0f` and

127 must convert to `1.0f`

For `CL_SNORM_INT16`

-32768 and -32767 must convert to `-1.0f`,

0 must convert to `0.0f` and

32767 must convert to `1.0f`

Converting floating-point values to normalized integer channel data types

For images created with image channel data type of `CL_UNORM_INT8` and `CL_UNORM_INT16`, **write_imagef** will convert the floating-point color value to an 8-bit or 16-bit unsigned integer.

For images created with image channel data type of `CL_SNORM_INT8` and `CL_SNORM_INT16`, **write_imagef** will convert the floating-point color value to an 8-bit or 16-bit signed integer.

The preferred method for how conversions from floating-point values to normalized integer values are performed is as follows:

`float` → `CL_UNORM_INT8` (8-bit unsigned integer)

convert_uchar_sat_rte(`f * 255.0f`)

`float` → `CL_UNORM_INT_101010` (10-bit unsigned integer)

min(**convert_ushort_sat_rte**(`f * 1023.0f`), `0x3ff`)

`float` → `CL_UNORM_INT16` (16-bit unsigned integer)

convert_ushort_sat_rte(`f * 65535.0f`)

`float` → `CL_SNORM_INT8` (8-bit signed integer)

convert_char_sat_rte(`f * 127.0f`)

`float` → `CL_SNORM_INT16` (16-bit signed integer)

convert_short_sat_rte(`f * 32767.0f`)

Please refer to the out-of-range behavior and saturated conversion rules.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described above. If a rounding mode other than round to nearest even (`_rte`) is used, the absolute error of the implementation dependant rounding mode vs. the result produced by the round to nearest even rounding mode must be ≤ 0.6.

`float` → `CL_UNORM_INT8` (8-bit unsigned integer)

Let $f_{preferred}$ = **convert_uchar_sat_rte**(f * `255.0f`)

Let f_{approx} = **convert_uchar_sat_<impl-rounding-mode>**(f * `255.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT_101010` (10-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `1023.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `1023.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT16` (16-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `65535.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `65535.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_SNORM_INT8` (8-bit signed integer)

Let $f_{preferred}$ = **convert_char_sat_rte**(f * `127.0f`)

Let f_{approx} = **convert_char_sat_<impl_rounding_mode>**(f * `127.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_SNORM_INT16` (16-bit signed integer)

Let $f_{preferred}$ = **convert_short_sat_rte**(f * `32767.0f`)

Let f_{approx} = **convert_short_sat_<impl-rounding-mode>**(f * `32767.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

8.3.2. Conversion rules for half precision floating-point channel data type

For images created with a channel data type of `CL_HALF_FLOAT`, the conversions from `half` to `float` are lossless (as described in The half data type). Conversions from `float` to `half` round the mantissa using the round to nearest even or round to zero rounding mode. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` may be flushed to zero. A `float` NaN must be converted to an appropriate NaN in the `half` type. A `float` `INF` must be converted to an appropriate `INF` in the `half` type.

8.3.3. Conversion rules for floating-point channel data type

The following rules apply for reading and writing images created with channel data type of `CL_FLOAT`.

- NaNs may be converted to a NaN value(s) supported by the device.
- Denorms can be flushed to zero.
- All other values must be preserved.

8.3.4. Conversion rules for signed and unsigned 8-bit, 16-bit and 32-bit integer channel data types

Calls to **read_imagei** with channel data type values of `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32` return the unmodified integer values stored in the image at specified location.

Calls to **read_imageui** with channel data type values of `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32` return the unmodified integer values stored in the image at specified location.

Calls to **write_imagei** will perform one of the following conversions:

32 bit signed integer → 8-bit signed integer

convert_char_sat(i)

32 bit signed integer → 16-bit signed integer

convert_short_sat(i)

32 bit signed integer → 32-bit signed integer

no conversion is performed

Calls to **write_imageui** will perform one of the following conversions:

32 bit unsigned integer → 8-bit unsigned integer

convert_uchar_sat(i)

32 bit unsigned integer → 16-bit unsigned integer

convert_ushort_sat(i)

32 bit unsigned integer → 32-bit unsigned integer

no conversion is performed

The conversions described in this section must be correctly saturated.

8.3.5. Conversion rules for sRGBA and sBGRA images

Standard RGB data, which roughly displays colors in a linear ramp of luminosity levels such that an average observer, under average viewing conditions, can view them as perceptually equal steps on an average display. All 0's maps to `0.0f`, and all 1's maps to `1.0f`. The sequence of unsigned integer encodings between all 0's and all 1's represent a nonlinear progression in the floating-point interpretation of the numbers between `0.0f` to `1.0f`. For more detail, see the SRGB color standard.

Conversion from sRGB space is automatically done by **read_imagef** built-in functions if the image channel order is one of the sRGB values described above. When reading from an sRGB image, the conversion from sRGB to linear RGB is performed before the filter specified in the sampler specified to read_imagef is applied. If the format has an alpha channel, the alpha data is stored in linear color space. Conversion to sRGB space is automatically done by **write_imagef** built-in functions if the image channel order is one of the sRGB values described above and the device supports writing to sRGB images.

If the format has an alpha channel, the alpha data is stored in linear color space.

The following is the conversion rule for converting a normalized 8-bit unsigned integer sRGB color value to a floating-point linear RGB color value using **read_imagef**.

```
// Convert the normalized 8-bit unsigned integer R, G and B channel values
// to a floating-point value (call it c) as per rules described in section
// 8.3.1.1.

if (c <= 0.04045),
    result = c / 12.92;
else
    result = powr((c + 0.055) / 1.055, 2.4);
```

The resulting floating point value, if converted back to an sRGB value without rounding to a 8-bit unsigned integer value, must be within 0.5 ulp of the original sRGB value.

The following are the conversion rules for converting a linear RGB floating-point color value (call it *c*) to a normalized 8-bit unsigned integer sRGB value using **write_imagef**.

```
if (c is NaN)
    c = 0.0;
if (c > 1.0)
    c = 1.0;
else if (c < 0.0)
    c = 0.0;
else if (c < 0.0031308)
    c = 12.92 * c;
else
    c = 1.055 * powr(c, 1.0/2.4) - 0.055;

scaled_reference_result = c * 255
channel_component = floor(scaled_reference_result + 0.5);
```

The precision of the above conversion should be such that

`|generated_channel_component - scaled_reference_result|` ≤ 0.6

where `generated_channel_component` is the actual value that the implementation produces and being checked for conformance.

8.4. Selecting an Image from an Image Array

Let (u,v,w) represent the unnormalized image coordinate values for reading from and/or writing to a 2D image in a 2D image array.

When read using a sampler, the 2D image layer selected is computed as:

layer = **clamp**(**rint**(w), 0, d_t - 1)

otherwise the layer selected is computed as:

layer = w

(since w is already an integer) and the result is undefined if w is not one of the integers 0, 1, ... d_t - 1.

Let (u,v) represent the unnormalized image coordinate values for reading from and/or writing to a 1D image in a 1D image array.

When read using a sampler, the 1D image layer selected is computed as:

layer = **clamp**(**rint**(v), 0, h_t - 1)

otherwise the layer selected is computed as:

layer = v

(since v is already an integer) and the result is undefined if v is not one of the integers 0, 1, ... h_t - 1.

Chapter 9. Normative References

1. “ISO/IEC 9899:1999 - Programming Languages - C”, with technical corrigenda TC1 and TC2, https://www.iso.org/standard/29237.html . References are to sections of this specific version, referred to as the “C99 Specification”, although other versions exist.
2. “ISO/IEC 9899:2011 - Information technology - Programming languages - C”, https://www.iso.org/standard/57853.html . References are to sections of this specific version, referred to as the “C11 Specification”, although other versions exist.
3. “The OpenCL Specification, Version 2.0”, https://www.khronos.org/registry/OpenCL/ . References are to sections and tables of this specific version, although other versions exists.
4. “Device Queries” are defined in the OpenCL Specification for **clGetDeviceInfo**, and the individual queries are defined in the “OpenCL Device Queries” table (4.3) of that Specification.
5. “Image Channel Order” is defined in the OpenCL Specification in the “Image Format Descriptor” section (5.3.1.1), and the individual channel orders are defined in the “List of supported Image Channel Order Values” table (5.6) of that Specification.
6. “Image Channel Data Type" is defined in the OpenCL Specification in the "`Image Format Descriptor” section (5.3.1.1), and the individual channel data types are defined in the "`List of supported Image Channel Data Types" table (5.7) of that Specification.
7. “The OpenCL Extension Specification, Version 2.0”, https://www.khronos.org/registry/OpenCL/ . References are to sections and tables of this specific version, although other versions exists.
8. “IEC 61966-2-1:1999 Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB”, https://webstore.iec.ch/publication/6169 .